Android 源码分析实录

基于当今市面主流版本
全面讲解Android系统的整体架构
细致剖析14大核心系统
底层、HAL层和应用层学习循序渐进

李忠良 编著

讲解详尽，深入底层

本书详细讲解了**Android源码分析**的每一个知识点，为了更加透彻地说明原理，从深入底层着手，到顶层Java应用结束，即使菜鸟也能够看懂并掌握。

大话模式，趣味性强

全书采用诙谐、生动的**大话模式讲解实例**，在逼真的生活场景中学习编程，将复杂的高深专业知识，以趣味性的语言讲解出来。

实例典型，提示丰富

书中的实例典型，融合了技术中所有的**经典范例**，以加深读者对知识的掌握。

高深内容详细剖析，做到一应俱全

作为某项专业技术，用最合理的篇幅**详细剖析了每个知识点**，内容涉及了领域内的方方面面，可直接作为此领域的权威书籍。

清华大学出版社

Android 源码分析实录

李忠良　编著

清华大学出版社
北　京

内 容 简 介

Android 是一款服务于智能手机和平板电脑等设备的操作系统，截止作者撰写此书时为止，Android 在智能手机操作系统市场中已经占有 75%的份额。为了让广大读者充分了解这款神奇的操作系统的架构原理，本书循序渐进地分析了 Android 系统核心源码的基本知识。

本书共分为 15 章，主要内容包括走进 Android 世界、硬件抽象层详解、分析 JNI(Java 本地接口)层、Android 内存系统分析、Android 虚拟机系统详解、IPC 通信机制详解、Zygote 进程/System 进程和应用程序进程、分析 Activity 组件、Content Provider 数据存储、Broadcast(广播)系统详解、多媒体系统详解、电源管理系统详解、输入系统驱动应用、蓝牙系统详解、网络系统详解等。

本书几乎涵盖了 Android 源码中的所有核心系统的内容，全书内容通俗易懂，适合 Android 初学者、Android 爱好者、Android 底层开发人员、Android 应用开发人员阅读和学习，也可以作为相关培训学校和大专院校相关专业的教学用书。

图书在版编目(CIP)数据

Android 源码分析实录/李忠良编著. --北京：清华大学出版社，2015
ISBN 978-7-302-39329-0

Ⅰ. ①A…　Ⅱ. ①李…　Ⅲ. ①移动终端—应用程序—程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2015)第 024952 号

责任编辑：杨作梅
封面设计：杨玉兰
责任校对：马素伟
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　张：**46.25　　**字　数：**1124 千字
版　　次：2015 年 4 月第 1 版　　**印　次：**2015 年 4 月第 1 次印刷
印　　数：1～3000
定　　价：89.00 元

产品编号：054375-01

前　　言

Android(中文译名为安卓)是 IT 界巨头 Google(谷歌)公司于 2007 年 11 月 5 日推出的一款智能操作系统，最初被应用于智能手机，后来随着版本的更新和发展，也被广泛应用于平板电脑、智能电视、可穿戴设备和健康设备中。Android 是一款基于 Linux 平台的开源操作系统的名称，根据国际数据公司(IDC)公布的数据，Android 在智能手机操作系统中的市场占有率已经达到 75%。

高份额的市场占有率使得更多的开发人员把目光投入这款神奇的系统，很多初学者也纷纷涌入 Android 的学习行列中，配合这些需求，Android 的各种应用类图书不断涌现并广受欢迎。但美中不足的是，深入源码分析的书籍屈指可数。而源码分析正是通往 Android 殿堂、跻身为高手的阶梯。

为了让广大初学者可以对 Android 系统实现“灵与肉”的感知，而不是停留在抽象的原理和概念上，本书对 Android 系统的源码进行细致的分析，这样做的目的，是提炼出 Android 系统埋藏于深处的本质和精华的东西，以展示这款神奇的系统究竟是怎样实现的。

1．本书内容

Android 系统升级较快，有些代码变动很大。系统自 2007 年发布第一个版本 1.1 以来，截至 2013 年 7 月发布版本 4.3，中间一共存在十多个版本。但据官方统计，到 2013 年 5 月 5 日，占据前三位的版本分别是 Android 4.2，Android 4.1 和 Android 4.3，其实这三个版本的区别并不是很大，只是在某领域的细节上进行了更新。因此，在本书中，我们选择本书最初写作时的最新版本 Android 4.3 系统的实现。

本书共分 15 章，依次为走进 Android 世界、硬件抽象层详解、分析 JNI(Java 本地接口)层、Android 内存系统分析、Android 虚拟机系统详解、IPC 通信机制详解、Zygote 进程/System 进程和应用程序进程、分析 Activity 组件、Content Provider 数据存储、Broadcast(广播)系统详解、多媒体系统详解、电源管理系统详解、输入系统驱动应用、蓝牙系统详解、网络系统详解。

本书几乎涵盖 Android 源码中的所有核心系统的内容，全书通俗易懂，特别有利于初学者学习和消化。

2．本书特色

本书内容十分丰富，分析细致、全面。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要，有选择地阅读。

在内容的编写上，本书具有以下特色。

(1)　结构合理

从用户的实际需要出发，科学安排知识结构。全书详细地讲解与 Android 应用开发有关的源码，内容循序渐进，由浅入深。

(2)　易学易懂

本书条理清晰、语言简洁，可帮助读者快速掌握每个知识点，使读者既可以按照本书编排

的章节顺序进行学习，也可以根据自己的需求，对某一章节进行有针对性的学习。

(3) 实用性强

本书彻底摒弃枯燥的理论知识罗列，注重实用性和可操作性，通过细腻的笔法，逐步讲解各个知识点的基本知识。

(4) 内容全面

本书是如今市面上"内容最全的 Android 源码分析书"，无论是获取源码，还是各个常用、常见的模块系统，在本书中您都能找到解决问题的答案。

3．读者对象

本书适合下列人员阅读和学习：

- 初学 Android 编程的自学者。
- Android 源码分析人员。
- Android 底层开发人员。
- Android 系统开发人员。
- 相关培训机构的教师和学员。
- 从事 Android 开发的程序员。

4．作者支持

在编写此书的过程中，得到了清华大学出版社工作人员的大力支持，正是由于各位编辑的求实态度、耐心的工作和奉献精神，才使得本书能够快速出版。

另外也十分感谢我的家人在我写作的时候给予的巨大支持。

由于作者水平有限，本书的疏漏之处在所难免，恳请读者提出意见或建议，以便再版时修订并使之更臻完善。我们提供了售后支持 QQ(号码为 1727069718)，读者如有疑问可以通过 QQ 提出，将会得到满意的答复。

编　者

目 录

第 1 章
走进Android世界

Android 系统于 2007 年诞生，是一款建立在 Linux 内核之上的智能设备系统，是一款经典的手机、平板电脑等移动设备的软件解决方案。从 2011 年下半年开始到现在，Android 系统在全球智能手机操作系统中的占有率一直位居第一。

本章将简单介绍 Android 系统的发展历程和背景，让读者了解 Android 系统的发展之路，充分体验这款无与伦比的操作系统。

1.1 Android系统的优势

为什么 Google 的 Android(安卓)系统能够在短短 4 年内超越了 Symbian(塞班)、Blackberry (黑莓)、iOS 等前辈，从一名后起之秀变为移动智能设备市场占有率的大佬？这需要从 Android 系统的优势谈起，在本节的内容中，将为读者展示这些优势。

1.1.1 开源

Google 的 Android 出身于 Linux 世家，是一款开源的手机操作系统。正因为如此，在 Android 崭露头角之后，各大手机厂商和电信部门纷纷加入到了 Android 联盟中。这个联盟由业界内的公认大佬组成，主要成员包括 Google、中国移动、摩托罗拉、高通和 T-Mobile 等在内的 30 多家技术和无线应用的领军企业。Android 通过与运营商、设备制造商、开发商和其他有关各方结成深层次的合作伙伴关系，希望通过建立标准化、开放式的移动电话软件平台，在移动产业内形成一个开放式的生态系统。

开源意味着对开发人员和手机厂商来说，Android 是完全无偿免费使用的。正是因为源代码公开的原因，所以吸引了全世界各地无数程序员的热情。于是很多手机厂商都纷纷采用 Android 作为自己产品的系统，这当然也包括很多山寨厂商。因为免费，所以降低了成本，因而提高了利润。而对于开发人员来说，因为 Android 被众多移动设备产品所采用，所以这方面的人才也变得愈发抢手。于是有一些在别的系统上干得还可以的程序员也改行做 Android 开发，纷纷加入到 Android 开发大军中来，原因是待遇更好；另外，也有很多混得不尽如人意的程序员更是纷纷改行做 Android 手机开发，目的是想寻找自己程序员生涯的转机。

而像本书作者这样遇到发展瓶颈的程序员，后来也决定做 Android 开发，因为这样可以学习一门新的技术，使自己的未来更加有保障。

1.1.2 强大的开发团队的支持

Android 的研发队伍阵容强大，包括 Google、摩托罗拉、HTC(宏达电子)、Philips、T-Mobile、高通、魅族、三星、LG 以及中国移动在内的 34 家企业，这些企业都基于 Android 平台开发手机的新型业务，并使应用之间的通用性和互联性在最大程度上得到保持。从硬件到软件开发机构，再到电信服务商，Android 从一开始便成为业界内的宠儿，被当作新秀而重点培养，在强大的开发团队的培育和呵护下，顺利地功成名就，成为一方霸主。

1.1.3 开发人员的支持

Google 一直视程序员为前进的动力和源泉，为了提高程序员们的开发积极性，不但为开发人员提供了一流的开发装备和软件服务，而且还提出了振奋人心的奖励机制。

具体的开发人员支持主要体现在如下三个方面。

(1) 可以迅速步入 Android 应用开发。在 Android 平台上，程序员可以开发出各式各样的应用。Android 应用程序是通过 Java 语言开发的，只要具备 Java 开发基础，就能很快地上手并掌握。对于单独的 Android 应用开发来说，并没有很高的 Java 编程门槛，即使没有编程经验的

门外汉，也可以在突击学习 Java 之后很快掌握 Android 编程。另外，Android 完全支持 2D、3D 和数据库，并且与浏览器实现了集成。所以，通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用，例如常见的工具、管理程序、互联网程序和游戏程序等。

(2) 可以参加奖金丰厚的 Android 大赛。为了吸引更多的用户使用 Android 开发，Google 定期举办奖金为数千万美元的开发者竞赛，鼓励开发人员做出创意十足的软件。这种大赛对于开发人员来说，不但能磨练自己的开发水平，并且高额奖金本身也是吸引学习的动力。

(3) 可以加入自由经营的贸易市场。为了能让 Android 平台吸引更多的关注，Google 提供了一个专门下载 Android 应用的门店：Android Market，网址是 https://play.google.com/store。在这个门店里面，允许开发人员发布应用程序，也允许 Android 用户下载自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且可以对自己的软件进行定价。只要你的软件程序足够吸引人，就可以获得很好的回报。学习和赚钱两不误，我们何乐而不为呢？

1.2　Android系统架构介绍

Android 是一个移动设备的开发平台，其软件层次结构大体上包括操作系统(OS)、中间件(Middleware)和应用程序(Application)。

Android 操作系统的组件结构如图 1-1 所示。

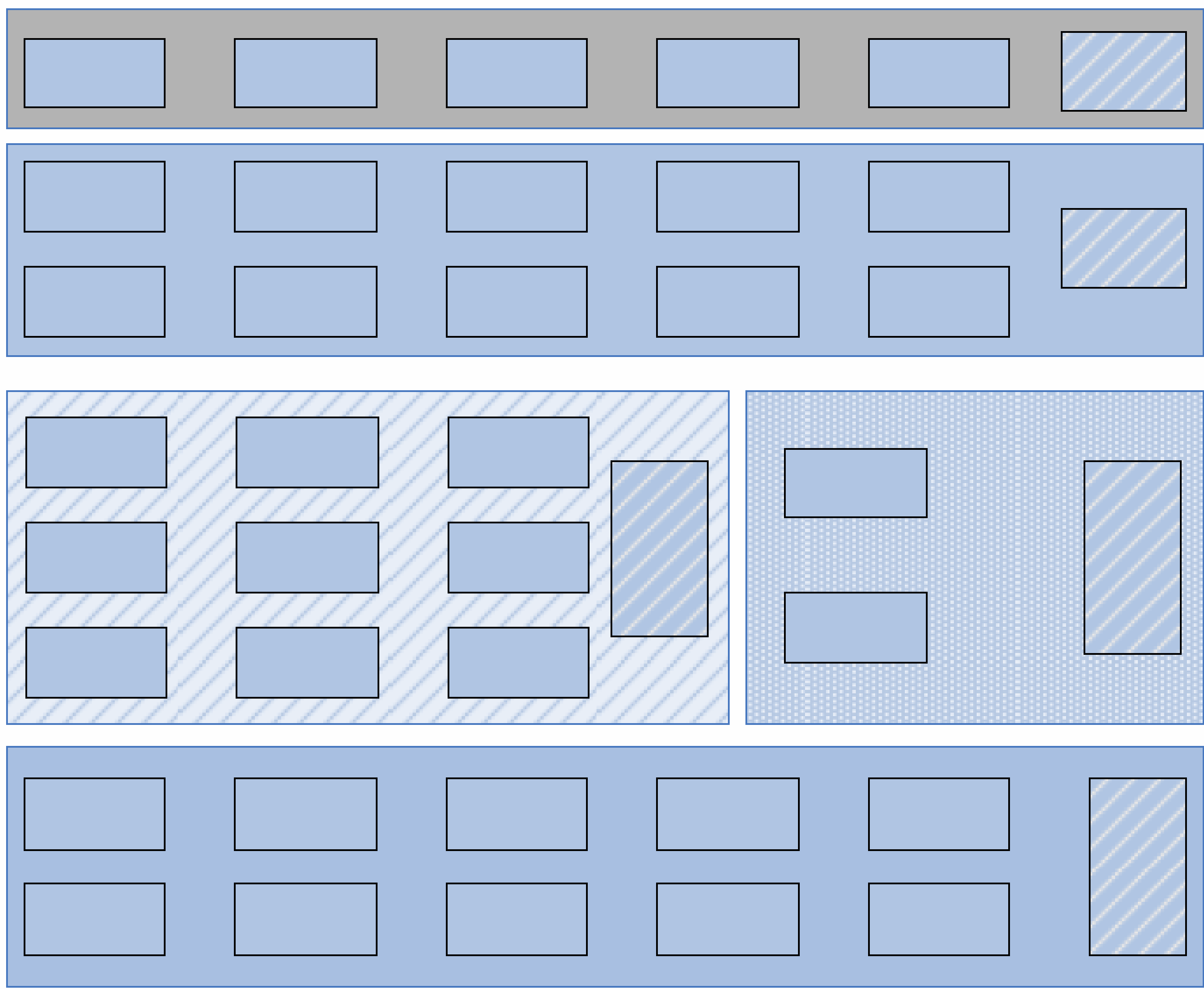


图 1-1　Android操作系统的组件结构

根据 Android 操作系统的组件结构框图可知，其软件层次结构自下而上分为 4 层。

(1) 操作系统层(即 Linux 内核层)。

(2) 各种库(Libraries)和 Android 运行环境(Runtime)。

(3) 应用程序框架(Application Framework)。

(4) 应用程序(Application)。

1.2.1 底层操作系统层(Linux内核层)

因为 Android 源于 Linux，使用了 Linux 内核，所以 Android 使用 Linux 2.6 作为操作系统。Linux 2.6 是一种标准的技术，Linux 也是一个开放的操作系统。Android 对操作系统的使用包括核心和驱动程序两部分，Android 的 Linux 核心为标准的 Linux 2.6 内核，Android 更多的是需要一些与移动设备相关的驱动程序。主要的驱动程序如下所示。

- 显示驱动(Display Driver)：是常用的基于 Linux 的帧缓冲(Frame Buffer)驱动程序。
- Flash 内存驱动(Flash Memory Driver)：是基于 MTD 的 Flash 驱动程序。
- 照相机驱动(Camera Driver)：常用基于 Linux 的 V4L(Video for Linux)驱动程序。
- 音频驱动(Audio Driver)：常用基于 ALSA(Advanced Linux Sound Architecture，高级 Linux 声音体系)的驱动程序。
- Wi-Fi 驱动(Wi-Fi Driver)：基于 IEEE 802.11 标准的驱动程序。
- 键盘驱动(Keyboard Driver)：作为输入设备的键盘驱动程序。
- 蓝牙驱动(Bluetooth Driver)：基于 IEEE 802.15.1 标准的无线传输技术驱动程序。
- Binder IPC 驱动：具有单独的设备节点，提供进程间通信功能的特殊驱动程序。
- Power Management(电源管理)：用于管理电池电量等信息的驱动程序。

1.2.2 库(Libraries)和运行环境(Runtime)

本层次对应一般的嵌入式系统，相当于中间件层次。Android 的本层次分成两个部分，一个是各种库，另一个是 Android 运行环境。本层的内容大多是使用 C 语言实现的，其中包含了如下所示的各种库。

- C 库：C 语言的标准库，也是系统中一个最为底层的库，C 库是通过 Linux 的系统调用实现的。
- 多媒体框架(Media Framework)：这部分内容是 Android 多媒体的核心部分，基于 Packet Video(即 PV)的 Open core，从功能上看，本库分为两大部分，一部分是音频、视频的回放(Play Back)，另一部分是则是音视频的记录(Recorder)。
- SGL：2D 图像引擎。
- SSL：即 Secure Socket Layer，位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。
- OpenGL ES：提供了对 3D 的支持。
- 界面管理工具(Surface Management)：提供管理显示子系统等功能。
- SQLite：一个通用的嵌入式数据库。
- WebKit：网络浏览器的核心。
- FreeType：位图和矢量字体的功能。

在一般情况下，Android 的各种库是以系统中间件的形式提供的，它们的显著特点是与移动设备平台的应用密切相关。

另外，Android 的运行环境主要是基于 Dalvik(虚拟机)技术的。而 Dalvik 与一般的 Java 虚拟机(Java Virtual Machine，JVM)是有如下区别的。

- Java 虚拟机：执行的是 Java 标准的字节码(Bytecode)。
- Dalvik：执行的是 Dalvik 可执行格式(.dex)的文件。在执行的过程中，每一个应用程序即一个进程(Linux 的一个 Process)。

二者最大的区别在于，JVM 是基于栈的虚拟机(Stack-based)，而 Dalvik 是基于寄存器的虚拟机(Register-based)。显然，后者最大的好处在于可以依据硬件实现更大的优化，这更适合移动设备的特点。

1.2.3　应用程序框架(Application Framework)

在整个 Android 系统中，与应用开发最相关的是 Application Framework，在这一层，Android 为应用程序层的开发者提供了各种功能强大的 APIs，这实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的。在本层提供了程序中所需要的各种控件，例如：Views(视图组件)、List(列表)、Grid(栅格)、Text Box(文本框)、Button(按钮)，甚至还有一个嵌入式的 Web 浏览器。

一个基本的 Android 应用程序可以利用应用程序框架中的以下 5 个部分。

- Activity：活动。
- Broadcast Intent Receiver：广播意图接收者。
- Service：服务。
- Content Provider：内容提供者。
- Intent and Intent Filter：意图和意图过滤器。

1.2.4　顶层应用程序(Application)

Android 的应用程序主要是用户界面(User Interface)方面的，本层通常使用 Java 语言编写，其中还可以包含各种被放置在“res”目录中的资源文件。Java 程序和相关资源在经过编译后，会生成一个 APK 包。Android 本身提供了主屏幕(Home)、联系人(Contact)、电话(Phone)、浏览器(Browsers)等众多的核心应用。同时，应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序。这也是 Android 开源的巨大潜力的体现。

注意：　我们目的是分析 Android 系统的源码，因为涉及的知识面涵盖了所有上述 4 个层次，所以学习过程任重而道远。

1.3　核 心 组 件

一个典型的 Android 程序通常由 5 个组件组成，这 5 个组件构成了 Android 的核心功能。在分析 Android 4.3 源码之前，本节将详细讲解 Android 应用程序的核心组件的基本知识。

1.3.1　Activity的界面表现

Activities 是这 5 个组件中最常用的一个组件。程序中 Activity 通常的表现形式是一个单独

的界面(screen)。每个 Activity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。大多数程序有多个 Activity。例如，一个文本信息程序有这么几个界面：显示联系人列表界面、写信息界面、查看信息界面或者设置界面等。每个界面都是一个 Activity。切换到另一个界面就是载入一个新的 Activity。某些情况下，一个 Activity 可能会给前一个 Activity 返回值——例如，一个让用户选择相片的 Activity 会把选择到的相片返回给其调用者。

打开一个新界面后，前一个界面就被暂停，并放入历史栈中(界面切换历史栈)。使用者可以回溯前面已经打开的存放在历史栈中的界面。也可以从历史栈中删除没有界面价值的界面。Android 在历史栈中保留程序运行产生的所有界面：从第一个界面，到最后一个。

1.3.2 Intent和IntentFilters界面切换

Android 通过一个专门的 Intent 类来进行界面的切换。Intent 描述了程序想做什么(Intent 意为意图，目的，意向)。Intent 类还有一个相关类 IntentFilter。Intent 是一个请求，用来明确做什么事情，IntentFilter 则描述了一个 Activity(或下文的 IntentReceiver)能处理什么意图。显示某人联系信息的 Activity 使用了一个 IntentFilter，就是说，它知道如何处理应用到此人数据的 VIEW 操作。Activities 在文件 AndroidManifest.xml 中使用 IntentFilters。

通过解析 Intents 可以实现 Activity 的切换，我们可以使用 startActivity(myIntent)启用新的 Activity。系统会考察所有安装程序的 IntentFilters，然后找到与 myIntent 匹配最好的 IntentFilters 所对应的 Activity。这个新 Activity 能够接收 Intent 传来的消息，并因此被启用。解析 Intents 的过程发生在 startActivity 被实时调用时，这样做有如下两个好处：

- Activities 仅发出一个 Intent 请求，便能重用其他组件的功能。
- Activities 可以随时被替换为有等价 IntentFilter 的新 Activity。

1.3.3 Service服务

Service 是一个没有用户界面(UI)且长驻系统的代码，最常见的例子是媒体播放器从播放列表中播放歌曲。在媒体播放器程序中，可能有一个或多个 Activities 让用户选择播放的歌曲。然而在后台播放歌曲时无需 Activity 干涉，因为用户希望在音乐播放的同时能够切换到其他界面。既然这样，媒体播放器 Activity 需要通过 Context.startService()启动一个 Service，这个 Service 在后台运行以保持继续播放音乐。在媒体播放器被关闭之前，系统会保持音乐后台播放 Service 的正常运行。可以用 Context.bindService()方法连接到一个 Service 上(如果 Service 未运行的话，连接后还会启动它)，连接后就可以通过一个 Service 提供的接口与 Service 进行通话。对音乐 Service 来说，提供了暂停和重放等功能。

Android 系统将会尝试保留那些启动了的或者绑定了的服务进程，具体说明如下。

(1) 如果该服务正在进程的 onCreate()、onStart()或者 onDestroy()这些方法中执行，那么主进程将会成为一个前台进程，以确保此代码不会被停止。

(2) 如果服务已经开始，那么它的主进程的重要性会低于所有的可见进程，但是会高于不可见进程。由于只有少数几个进程是用户可见的，所以，只要不是内存特别低，该服务就不会停止。

(3) 如果有多个客户端绑定了服务，只要客户端中的一个对于用户是可见的，就可以认为该服务可见。

1.3.4　用Broadcast IntentReceiver广播

当要执行一些与外部事件相关的代码时，比如来电响铃时，或者到半夜时，就可能用到IntentReceiver。尽管 IntentReceivers 使用 NotificationManager 来通知用户一些好玩事情的发生，但没有用户界面。

IntentReceivers 可以在文件 AndroidManifest.xml 中声明，也可以用 Context.registerReceiver()来声明。当一个 IntentReceiver 被触发时，如果需要，系统自然会自动启动程序。程序也可以通过 Context.broadcastIntent()来发送自己的 Intent 广播给其他程序。

1.3.5　用Content Provider存储

在 Android 系统中，应用程序将数据存放在一个 SQLite 数据库格式的文件里，或者存放在其他有效设备里。如果想让其他程序能够使用我们程序中的数据，此时 Content Provider 就很有用了。Content Provider 是一个实现了一系列标准方法的类，这个类使得其他程序能存储、读取某种 Content Provider 可处理的数据。

1.4　进程和线程

在 Android 系统中，进程和线程用于完成某个 Android 任务。当第一次运行某个组件的时候，Android 会启动一个进程。在默认情况下，所有的组件和程序运行在这个进程和线程中，也可以安排组件在其他的进程或者线程中运行。在本节的内容中，将简要介绍 Android 系统中进程和线程的基本知识。

1.4.1　什么是进程

组件运行的进程是由 manifest 文件控制的。组件的节点一般都包含一个 process 属性，例如<activity>、<service>、<receiver>和<provider>节点。

属性 process 可以设置组件运行的进程，可以配置组件在一个独立进程中运行，或者多个组件在同一个进程中运行，甚至可以让多个程序在一个进程中运行，当然前提是这些程序共享一个 User ID 并给定同样的权限。另外，<application>节点也包含了 process 属性，用来设置程序中所有组件的默认进程。

当更加常用的进程无法获取足够的内存时，Android 会智能地关闭不常用的进程。当下次启动程序的时候，会重新启动这些进程。

当决定哪个进程需要被关闭的时候，Android 会考虑进程对用户是否还有用。例如 Android 会倾向于关闭一个长期不显示在界面的进程来支持一个经常显示在界面的进程。是否关闭一个进程，决定于组件在进程中的状态。

1.4.2　什么是线程

当用户界面需要快速对用户操作进行响应时，就需要将一些费时的操作，如网络连接、下载或者非常占用服务器时间的操作等放到其他线程中。也就是说，即使为组件分配了不同的进程，有时候也需要再分配线程。

线程是通过 Java 的标准对象 Thread 来创建的，在 Android 中提供了如下管理线程的方法。

(1)　Looper 可用来在线程中运行一个消息循环。

(2)　Handler 可用来传递一个消息。

(3)　HandlerThread 可用来创建一个带有消息循环的线程。

(4)　Android 让一个应用程序在单独的线程中，直到它创建自己的线程。

(5)　应用程序组件(Activity、Service、Broadcast Receiver)全都在理想的主线程中实例化。

(6)　当被系统调用时，任何组件都不应该执行长时间或是阻塞的操作(例如网络调用或是计算循环)，否则应该中断所有在该进程中的其他组件。

(7)　可以创建一个新的线程来执行长期操作。

1.5　获取Android 4.3 源码

在分析 Android 4.3 的源码之前，需要先获取其源码。因为目前市面上的主流操作系统是 Windows、Linux 和 Mac OS，由于 Mac OS 属于类 Linux 系统，所以本节将讲解在 Windows 系统和 Linux 系统中获取 Android 源码的知识。

1.5.1　在Linux系统中获取Android源码

在 Linux 系统中，通常使用 Ubuntu 来下载和编译 Android 源码。由于 Android 的源码内容很多，Google 采用了 git 的版本控制工具，并对不同的模块设置不同的 git 服务器，我们可以用 repo 自动化脚本来下载 Android 源码，下面介绍如何一步一步地获取 Android 源码的过程。

(1)　下载 repo

在用户目录下，创建 bin 文件夹，用于存放 repo，并把该路径设置到环境变量中去，命令如下：

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

下载 repo 的脚本，用于执行 repo，命令如下：

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
```

设置可执行权限，命令如下：

```
$ chmod a+x ~/bin/repo
```

(2)　初始化一个 repo 的客户端

在用户目录下，创建一个空目录，用于存放 Android 源码，命令如下：

```
$ mkdir AndroidCode
$ cd AndroidCode
```

进入到 AndroidCode 目录，并运行 repo 来下载源码，下载主线分支的代码，主线分支包括最新修改，以及并未正式发出版本的最新源码，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

下载其他分支，即正式发布的版本，可以通过添加-b 参数来下载，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest -b
android-4.2_r1
```

在下载过程中会需要填写 Name 和 E-mail，填写完毕之后，选择 Y 进行确认，最后提示 repo 初始化完成，这时可以开始同步 Android 源码了，同步过程很漫长，需要耐心等待，执行下面的命令开始同步代码：

```
$ repo sync
```

经过上述步骤后，便开始下载并同步 Android 源码了，界面效果如图 1-2 所示。

```
Checking out files: 100% (2497/2497), done. out files:  31% (790/2497)
Checking out files: 100% (1654/1654), done. out files:  39% (649/1654)
Checking out files: 100% (3471/3471), done.
Checking out files: 100% (24607/24607), done.ut files:  21% (5269/24607)
Checking out files: 100% (2431/2431), done. out files:  32% (800/2431)
Checking out files: 100% (18696/18696), done.
Checking out files: 100% (4276/4276), done. out files:  48% (2058/4276)
Checking out files: 100% (2216/2216), done. out files:  49% (1093/2216)
Checking out files: 100% (857/857), done.ng out files:  13% (115/857)
Checking out files: 100% (1141/1141), done. out files:   2% (28/1141)
Checking out files: 100% (431/431), done.ng out files:  10% (46/431)
Checking out files: 100% (175/175), done.ng out files:  10% (19/175)
Checking out files: 100% (135/135), done.
Checking out files: 100% (378/378), done.
Checking out files: 100% (433/433), done.
Checking out files: 100% (2407/2407), done. out files:  30% (724/2407)
Checking out files: 100% (2489/2489), done.
Checking out files: 100% (2493/2493), done.
Checking out files: 100% (177/177), done.ng out files:  15% (27/177)
Checking out files: 100% (137/137), done.
Checking out files: 100% (40775/40775), done.ut files:   5% (2199/40775)
Checking out files: 100% (93/93), done.
Checking out files: 100% (450/450), done.
Checking out files: 100% (5265/5265), done. out files:  41% (2167/5265)
Syncing work tree: 100% (329/329), done.
```

图 1-2　下载同步

1.5.2　在Windows平台上获取Android源码

Windows 平台上获取源码与 Linux 上面的原理相同，但是需要预先在 Windows 平台上搭建一个 Linux 环境，此处需要用到 Cygwin 工具。Cygwin 的作用是构建一套在 Windows 上的 Linux 模拟环境。下载 Cygwin 工具的地址如下：

```
http://cygwin.com/install.html
```

下载成功后，会得到一个名为“setup.exe”的可执行文件，通过此文件，可以更新和下载最新的工具版本，具体流程如下。

(1) 启动 Cygwin，如图 1-3 所示。

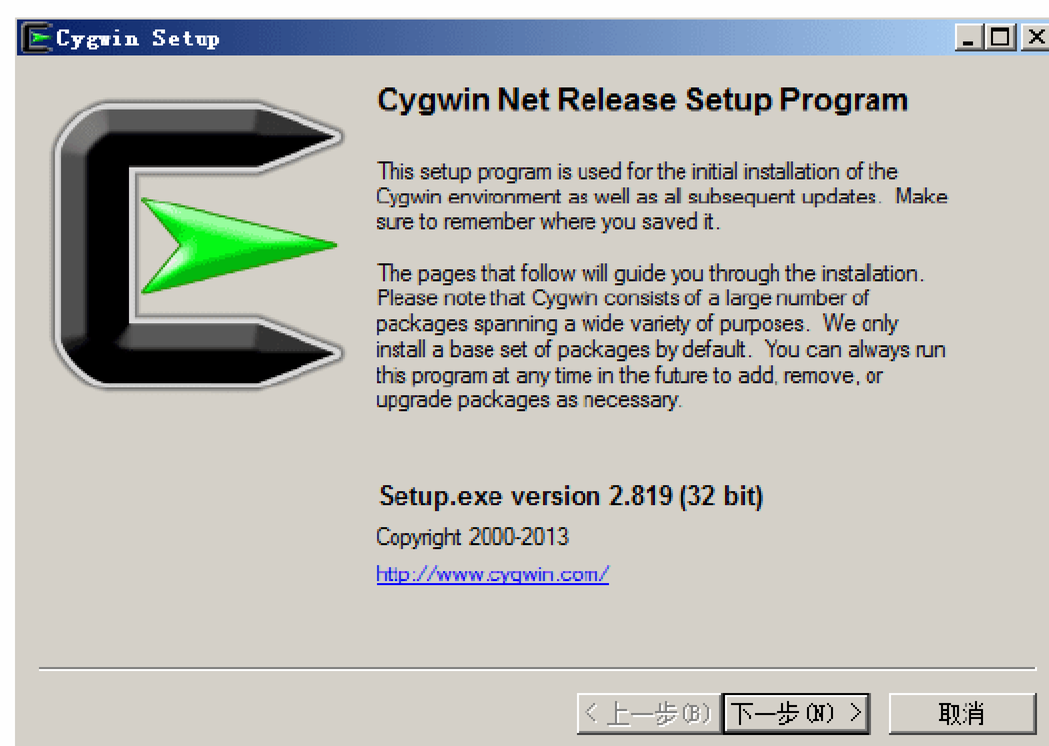


图 1-3 启动Cygwin

(2) 单击“下一步”按钮，选择第一个选项：从网络下载安装，如图 1-4 所示。

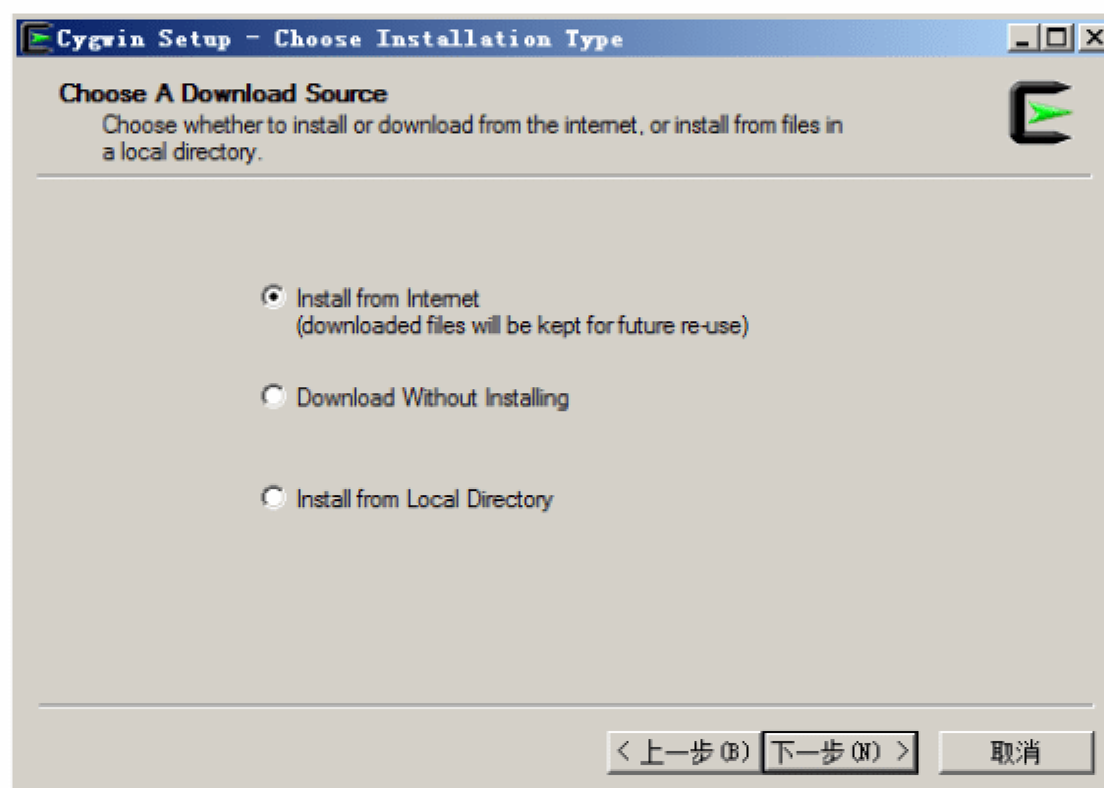


图 1-4 选择从网络下载安装

(3) 单击“下一步”按钮，选择安装根目录，如图 1-5 所示。

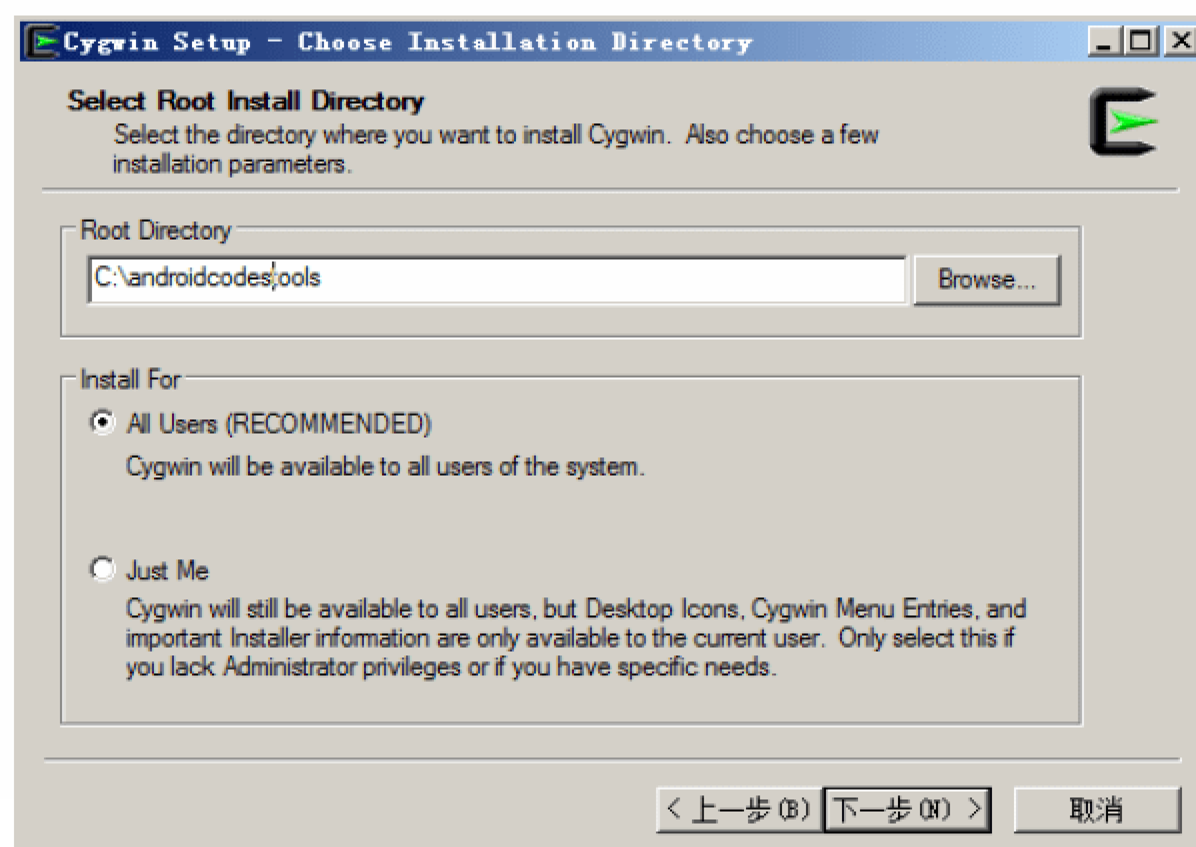


图 1-5 选择安装根目录

(4)　单击“下一步”按钮，选择临时文件目录，如图 1-6 所示。

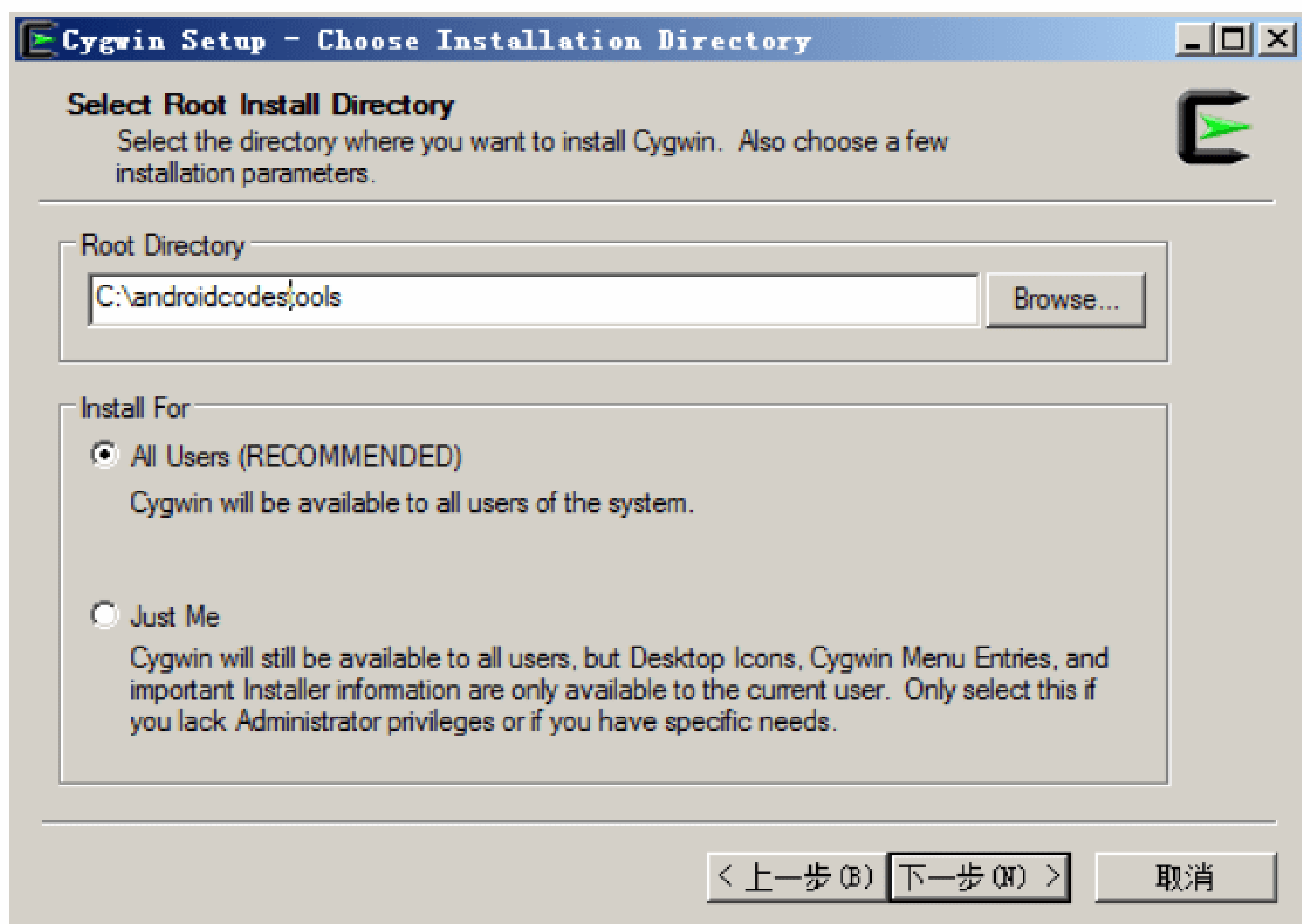


图 1-6　选择临时文件目录

(5)　单击“下一步”按钮，设置网络代理。如果所在网络需要代理，则在这一步进行设置，如果不用代理，则选择直接下载，如图 1-7 所示。

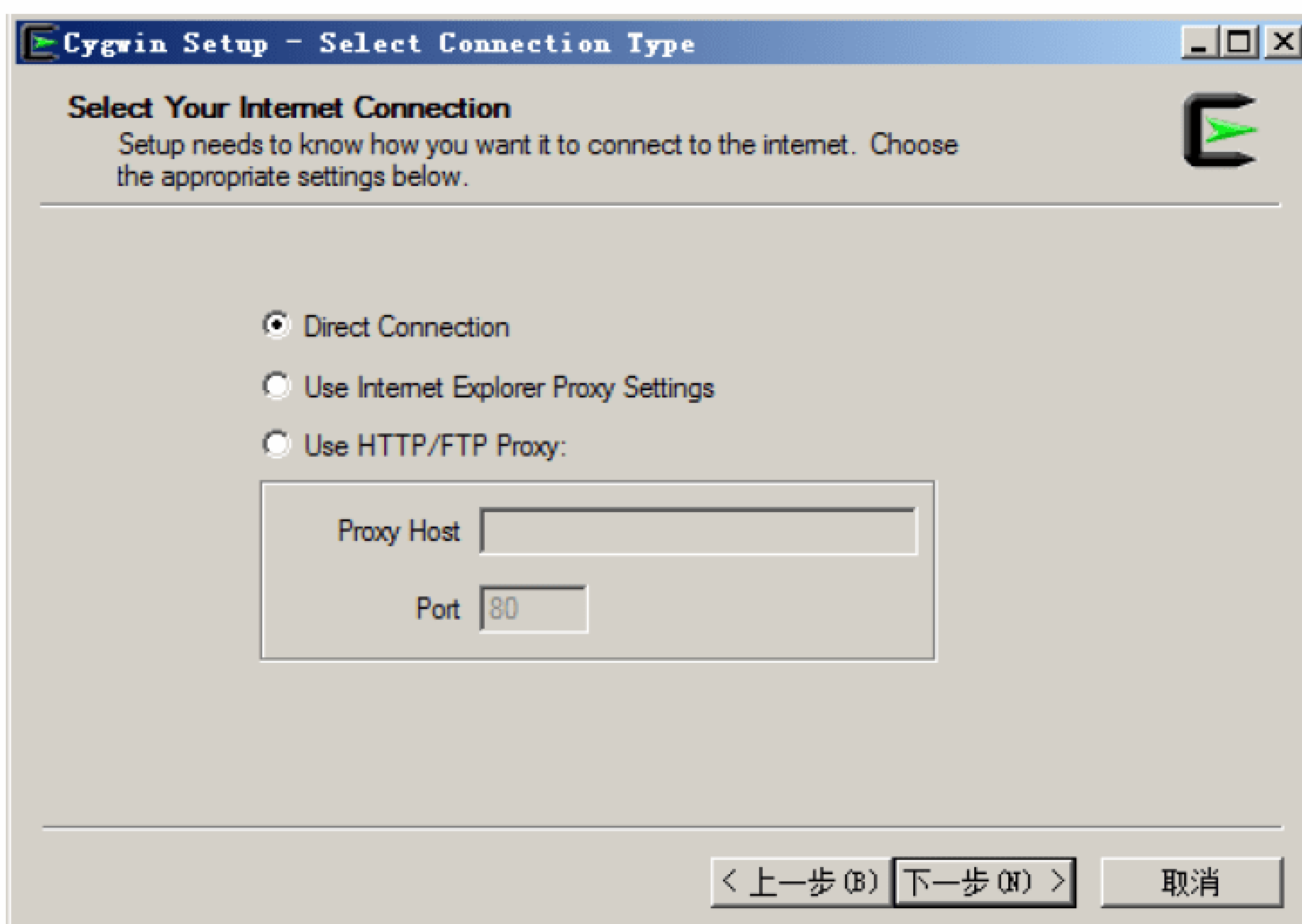


图 1-7　设置网络代理

(6)　单击“下一步”按钮，选择下载站点。一般选择离我们比较近的站点，速度会比较快，这里选择的是台湾省的站点，如图 1-8 所示。

(7)　单击“下一步”按钮，开始更新工具列表，如图 1-9 所示。

(8)　单击“下一步”按钮，选择需要下载的工具包。在此我们需要依次下载 curl、git、python 工具，如图 1-10 所示。

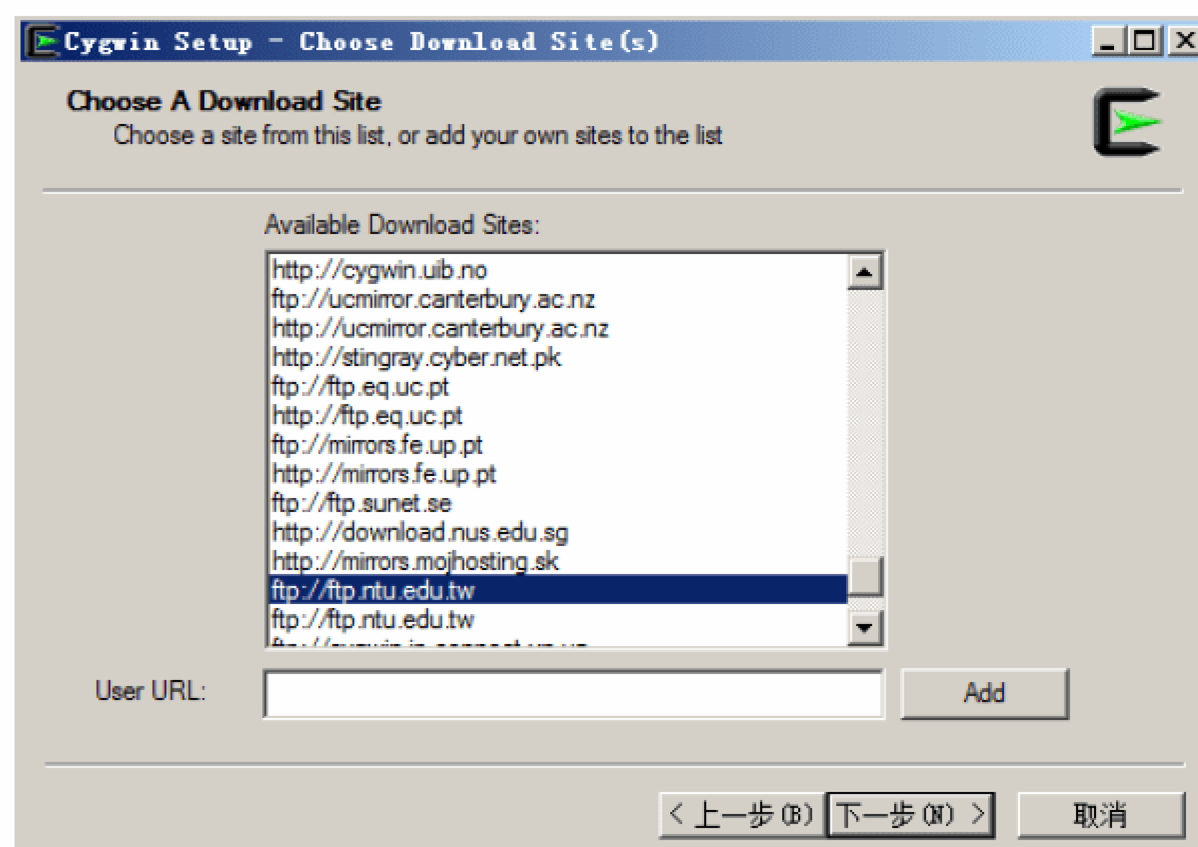


图 1-8　选择下载站点

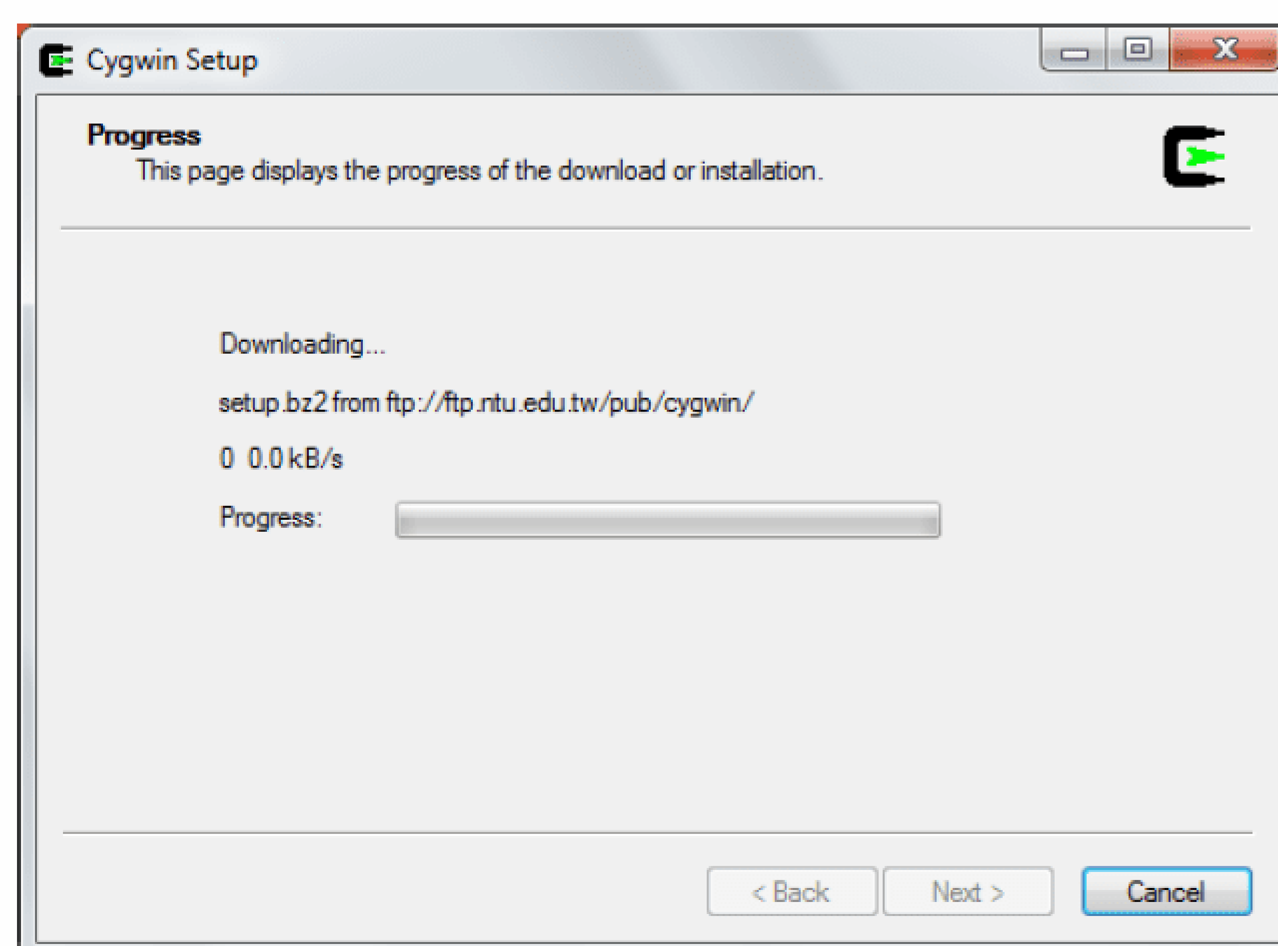


图 1-9　更新工具列表

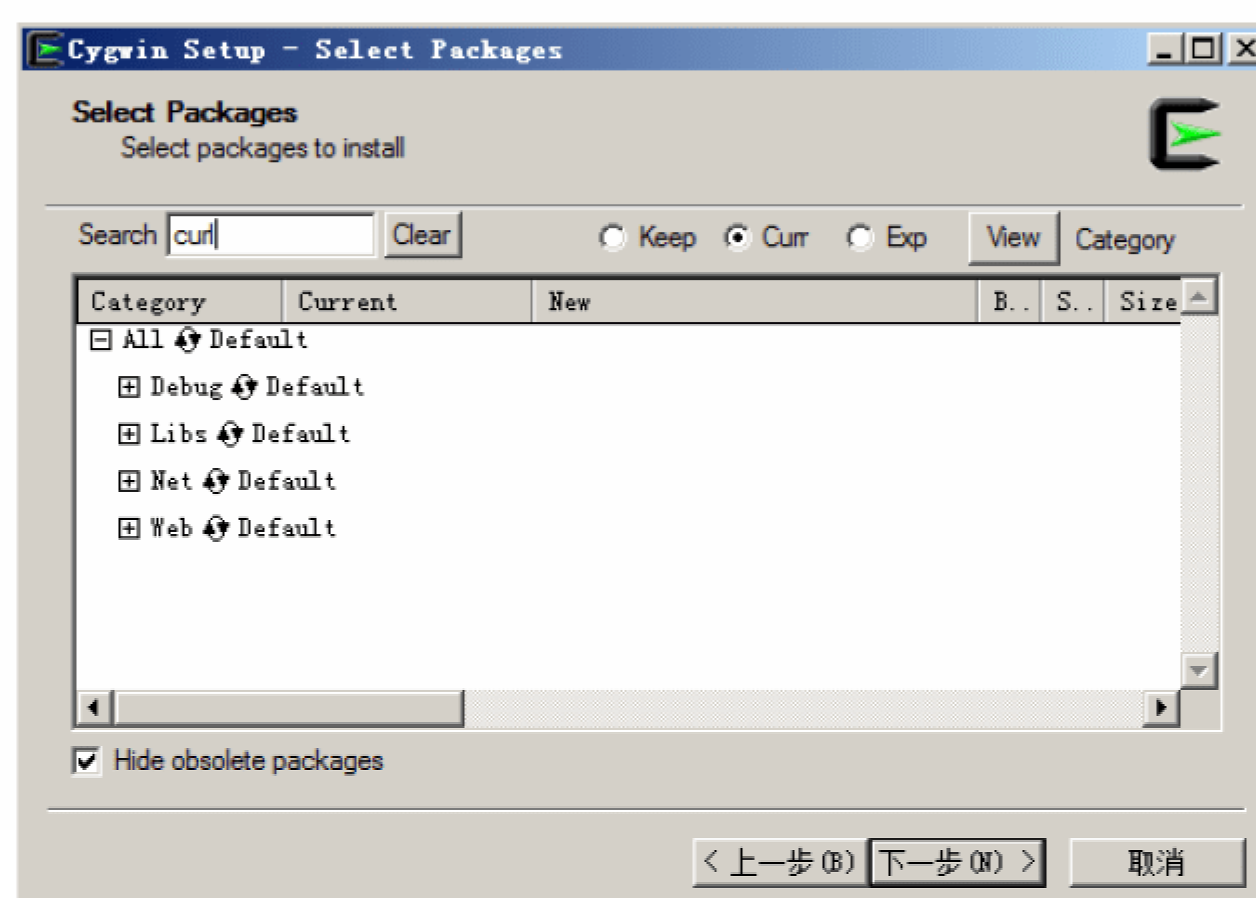


图 1-10　依次下载工具

为了确保能够安装上述工具，一定要用鼠标双击变为 Install 形式，如图 1-11 所示。

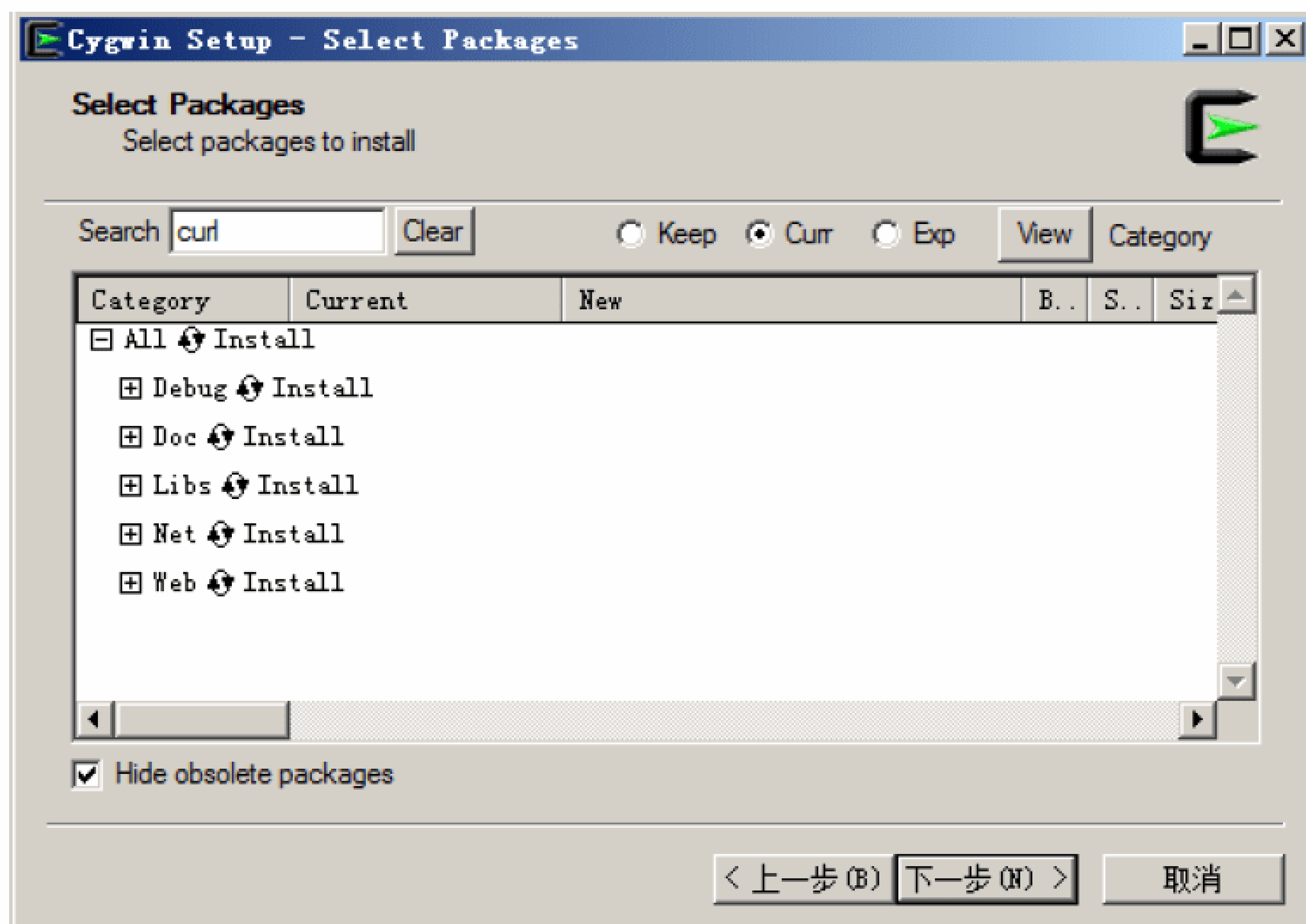


图 1-11　务必设置为Install形式

(9)　单击“下一步”按钮，需要经过漫长的等待过程，如图 1-12 所示。

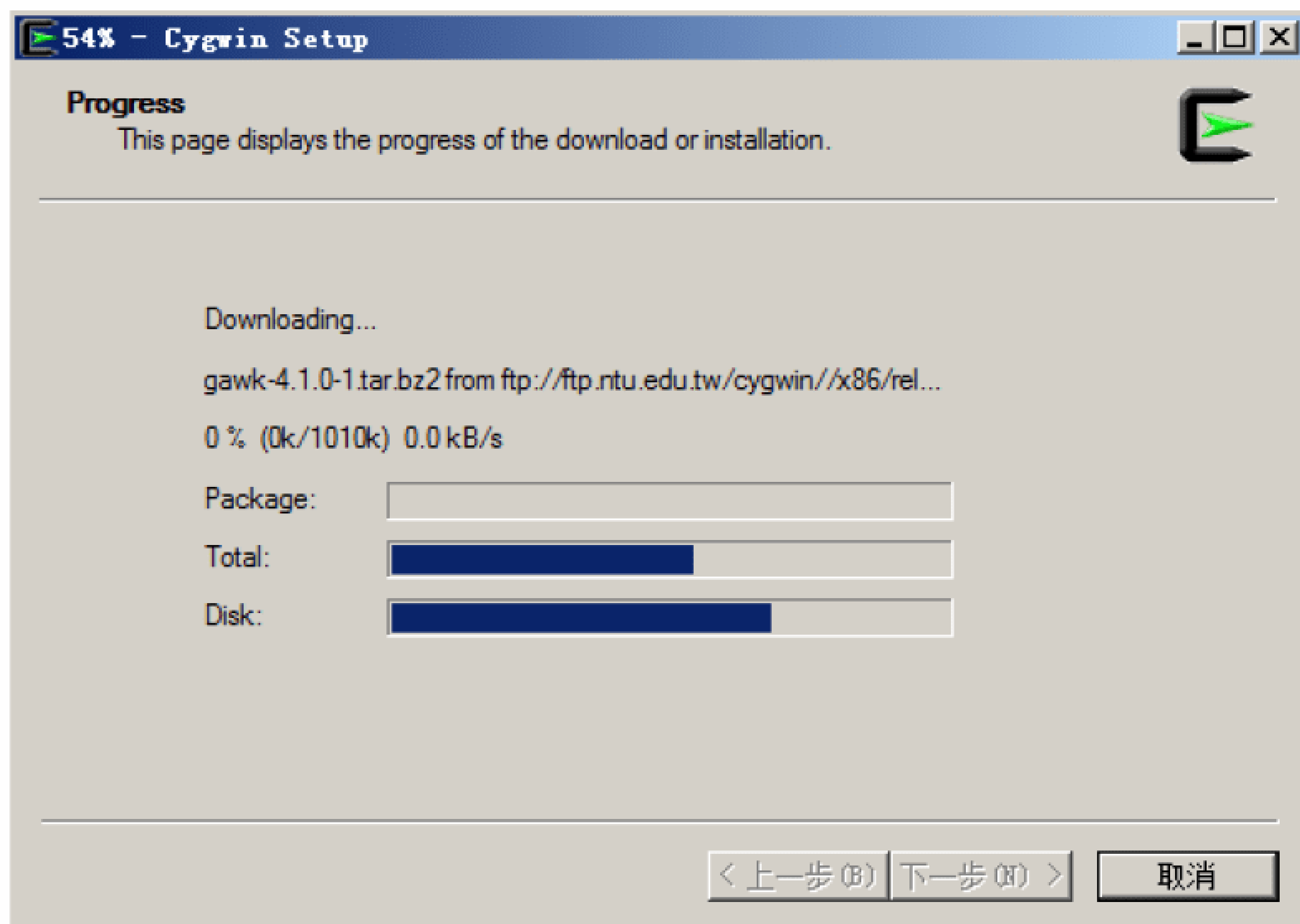


图 1-12　下载进度条

如果下载安装成功，会出现提示信息，单击“完成”按钮即完成安装。当安装好 Cygwin 后，打开 Cygwin，会模拟出一个 Linux 的工作环境，然后按照 Linux 平台的源码下载方法，就可以下载 Android 源码了。建议读者在下载 Android 源码时，严格按照官方提供的步骤进行，地址是 http://source.android.com/source/downloading.html，这一点对初学者来说尤为重要。另外，整个下载过程比较漫长，需要耐心地等待。图 1-13 是作者电脑上的命令截图。

```
~/WORKING_DIRECTORY
 * [new tag]         android-4.0.4_r1.1 -> android-4.0.4_r1.1
 * [new tag]         android-4.0.4_r1 -> android-4.0.4_r1
 * [new tag]         android-4.0.3_r1.1 -> android-4.0.3_r1.1
 * [new tag]         android-4.0.3_r1 -> android-4.0.3_r1
 * [new tag]         android-4.0.2_r1 -> android-4.0.2_r1
 * [new tag]         android-4.0.1_r1.2 -> android-4.0.1_r1.2
 * [new tag]         android-4.0.1_r1.1 -> android-4.0.1_r1.1
 * [new tag]         android-4.0.1_r1 -> android-4.0.1_r1
 * [new tag]         android-2.3_r1 -> android-2.3_r1
 * [new tag]         android-2.3.7_r1 -> android-2.3.7_r1
 * [new tag]         android-2.3.6_r1 -> android-2.3.6_r1
 * [new tag]         android-2.3.6_r0.9 -> android-2.3.6_r0.9
 * [new tag]         android-2.3.5_r1 -> android-2.3.5_r1
 * [new tag]         android-2.3.4_r1 -> android-2.3.4_r1
 * [new tag]         android-2.3.4_r0.9 -> android-2.3.4_r0.9
 * [new tag]         android-2.3.3_r1.1 -> android-2.3.3_r1.1
 * [new tag]         android-2.3.3_r1 -> android-2.3.3_r1
 * [new tag]         android-2.3.2_r1 -> android-2.3.2_r1
 * [new tag]         android-2.3.1_r1 -> android-2.3.1_r1
 * [new tag]         android-2.2_r1.3 -> android-2.2_r1.3
 * [new tag]         android-2.2_r1.2 -> android-2.2_r1.2
 * [new tag]         android-2.2_r1.1 -> android-2.2_r1.1
 * [new tag]         android-2.2_r1 -> android-2.2_r1
 * [new tag]         android-2.2.3_r2.1 -> android-2.2.3_r2.1
 * [new tag]         android-2.2.3_r2 -> android-2.2.3_r2
 * [new tag]         android-2.2.3_r1 -> android-2.2.3_r1
 * [new tag]         android-2.2.2_r1 -> android-2.2.2_r1
 * [new tag]         android-2.2.1_r2 -> android-2.2.1_r2
 * [new tag]         android-2.2.1_r1 -> android-2.2.1_r1
 * [new tag]         android-2.1_r2.1s -> android-2.1_r2.1s
 * [new tag]         android-2.1_r2.1p2 -> android-2.1_r2.1p2
 * [new tag]         android-2.1_r2.1p -> android-2.1_r2.1p
 * [new tag]         android-2.1_r2 -> android-2.1_r2
 * [new tag]         android-2.1_r1 -> android-2.1_r1
 * [new tag]         android-2.0_r1 -> android-2.0_r1
 * [new tag]         android-2.0.1_r1 -> android-2.0.1_r1
 * [new tag]         android-1.6_r2 -> android-1.6_r2
 * [new tag]         android-1.6_r1.5 -> android-1.6_r1.5
 * [new tag]         android-1.6_r1.4 -> android-1.6_r1.4
 * [new tag]         android-1.6_r1.3 -> android-1.6_r1.3
 * [new tag]         android-1.6_r1.2 -> android-1.6_r1.2
 * [new tag]         android-1.6_r1.1 -> android-1.6_r1.1
 * [new tag]         android-1.6_r1 -> android-1.6_r1
 24  154M   24 37.4M    0     0   161k      0  0:16:23  0:03:57  0:12:26  132k|:37  178k
```

图 1-13　在Windows中用Cygwin工具下载Android源码

1.6　Android源码结构分析

获得 Android 源码后，我们来分析源码的结构。源码的全部工程分为如下三个部分。

- Core Project：核心工程部分，这是建立 Android 系统的基础，保存在根目录的各个文件夹中。
- External Project：扩展工程部分，可以使其他开源项目具有扩展功能，保存在 external 文件夹中。
- Package：包部分，提供了 Android 的应用程序、内容提供者、输入法和服务，保存在 package 文件夹中。

在获取的 Android 4.3 源码目录中，包含了原始 Android 的目标机代码、主机编译工具和仿真环境。解压缩下载的 Android 4.3 源码包后，第一级别目录结构的具体说明如表 1-1 所示。

表 1-1　Android 4.3 源码的根目录

根 目 录	描　述
abi	abi 相关代码，应用程序二进制接口
bionic	bionic C 库
bootable	启动引导相关代码

续表

根 目 录	描　述
build	存放系统编译规则及 generic 等基础开发配置包
cts	Android 兼容性测试套件标准
dalvik	dalvik Java 虚拟机
development	应用程序开发相关代码
device	设备相关代码
docs	介绍开源的相关文档
external	Android 使用的一些开源的模组
frameworks	核心框架——Java 及 C++语言，是 Android 应用程序的框架
gdk	即时通信模块
hardware	主要是硬件适配层 HAL 代码
kernel	Linux 的内核文件
libcore	核心库相关
libnativehelper	是 Support functions for Android’s class libraries 的别名，表示动态库，是实现 JNI 库的基础
ndk	ndk 相关代码。Android NDK(Android Native Development Kit)是一系列的开发工具，允许程序开发人员在 Android 应用程序中嵌入 C/C++语言编写的非托管代码
out	编译完成后的代码输出在此目录中
packages	应用程序包
pdk	Plug Development Kit 的缩写，是本地开发套件
prebuilts	x86 和 ARM 架构下预编译的一些资源
sdk	SDK 及模拟器
system	文件系统和应用及组件，是用 C 语言实现的
tools	工具文件夹
vendor	厂商定制代码
Makefile	全局的 Makefile

在接下来的内容中，将详细讲解 Android 4.3 源码的基本结构。

1.6.1　Android源码的目录结构

当下载好 Android 4.3 的源码后，可以看到，第一级目录有 18 个文件夹和一个 Makefile 文件，如果是编译后的源码目录，会增加一个 out 文件夹，用来存放编译产生的文件，下面具体来分析一下这些目录各自的作用：

```
├── abi              //应用程序的二进制接口
├── bionic           //Android 基础 C 库的源码
├── bootable          //系统启动器的源码
├── build            //编译和配置系统所需要的配置文件和脚本文件
├── cts              //Android 兼容性测试标准
```

```
├── dalvik        //Android 虚拟机源码
├── development  //程序开发的模板和工具
├── device        //设备相关代码
├── docs          //开源的相关文档
├── external      //Android 使用的第三方开源库的源码
├── frameworks    //应用程序框架源码
├── gdk           //即时通信模块
├── hardware      //硬件抽象层源码
├── libcore       //相关核心库的代码
├── libnativehelper   //动态库
├── ndk           //NDK 开发环境
├── packages      //应用程序包
├── pdk           //本地开发套件
├── prebuilt      //x86 和 ARM 架构下预编译的一些资源
├── sdk           //SDK 和模拟器相关代码
├── system        //文件系统、应用和组件
├── Makefile      //系统编译脚本
```

通过上面对源码根目录中的每个文件夹的介绍，可以看出源码是按照功能进行分类的，整个 Android 源码分为系统代码、工具、文档、开发环境、虚拟机、配置脚本和编译脚本等类别。

1.6.2 应用程序

在 Android 源码中，应用程序部分的功能是实现 UI 界面，开发人员基于 SDK 开发的 APK 包便属于应用程序层。应用程序层在 Android 系统中处于最顶层，Android 4.3 源码结构中的 packages 目录用来实现系统的应用程序，此目录的具体结构如下所示：

```
packages /
├── apps //应用程序库
|    ├── BasicSmsReceiver         //基础短信接收
|    ├── Bluetooth               //蓝牙
|    ├── Browser                 //浏览器
|    ├── Calculator              //计算器
|    ├── Calendar                //日历
|    ├── Camera                  //照相机
|    ├── CellBroadcastReceiver  //单元广播接收
|    ├── CertInstaller           //被调用的包，在 Android 中安装数字签名
|    ├── Contacts                //联系人
|    ├── DeskClock               //桌面时钟
|    ├── Email                   //电子邮件
|    ├── Exchange                //Exchange 服务
|    ├── Gallery                 //图库
|    ├── Gallery2                //图库 2
|    ├── HTMLViewer              //HTML 查看器
|    ├── KeyChain                //密码管理
|    ├── Launcher2               //启动器 2
|    ├── Mms                     //彩信
|    ├── Music                   //音乐
```

```
|     ├── MusicFX                  //音频增强
|     ├── Nfc                      //近场通信
|     ├── PackageInstaller         //包安装器
|     ├── Phone                    //电话
|     ├── Protips                  //主屏幕提示
|     ├── Provision                //引导设置
|     ├── QuickSearchBox           //快速搜索框
|     ├── Settings                 //设置
|     ├── SoundRecorder            //录音机
|     ├── SpareParts               //系统设置
|     ├── SpeechRecorder           //录音程序
|     ├── Stk                      //sim 卡相关
|     ├── Tag                      //标签
|     ├── VideoEditor              //视频编辑
|     └── VoiceDialer              //语音编号
├── experimental                   //非官方的应用程序
|     ├── BugReportSender          //Bug 的报告程序
|     ├── Bummer
|     ├── CameraPreviewTest        //照相机预览测试程序
|     ├── DreamTheater
|     ├── ExampleImsFramework
|     ├── LoaderApp
|     ├── NotificationLog
|     ├── NotificationShowcase
|     ├── procstatlog
|     ├── RpcPerformance
|     └── StrictModeTest
├── inputmethods                   //输入法
|     ├── LatinIME                 //拉丁文输入法
|     ├── OpenWnn                  //OpenWnn 输入法
|     └── PinyinIME                //拼音输入法
├── providers                      //提供器
|     ├── ApplicationsProvider     //应用程序提供器，提供应用程序所需的界面
|     ├── CalendarProvider         //日历提供器
|     ├── ContactsProvider         //联系人提供器
|     ├── DownloadProvider         //下载管理提供器
|     ├── DrmProvider              //数据库相关
|     ├── GoogleContactsProvider   //Google 联系人提供器
|     ├── MediaProvider            //媒体提供器
|     ├── TelephonyProvider        //彩信提供器
|     └── UserDictionaryProvider   //用户字典提供器
├── screensavers                   //屏幕保护
|     ├── Basic                    //基本屏幕保护
|     ├── PhotoTable               //照片方格
|     ├── WebView                  //网页
└── wallpapers                     //墙纸
      ├── Basic                    //系统内置墙纸
      ├── Galaxy4                  //S4 内置墙纸
      ├── HoloSpiral               //手枪皮套墙纸
```

```
├── LivePicker
├── MagicSmoke
├── MusicVisualization
├── NoiseField
└── PhaseBeam
```

通过上面的目录结构可以看出，在 package 目录中包含了应用程序相关的包或者资源文件，不但包括系统自带的应用程序，也包括第三方开发的应用程序和屏幕保护和墙纸等应用。

1.6.3 应用程序框架

应用程序框架是 Android 系统中的核心部分，也就是 SDK 部分，它会提供接口给应用程序使用，同时应用程序框架又会与系统服务、系统程序库、硬件抽象层有关联，所以其作用十分重大，应用程序框架的实现代码大部分都在/frameworks/base 和/frameworks/av 目录下。

frameworks/base 的目录结构如下所示：

```
frameworks/base
├── api        //全是 XML 文件，定义了 API
├── cmds       //Android 中的重要命令(am、app_proce 等)
├── core       //核心库
├── data       //声音字体等数据文件
├── docs       //文档
├── drm        //数字版权管理
├── graphics   //图形图像
├── icu4j      //用于解决国际化问题
├── include    //头文件
├── keystore   //数字签名证书相关
├── libs       //库
├── location   //地理位置
├── media      //多媒体
├── native     //本地库
├── nfc-extras //NFC 相关
├── obex       //蓝牙传输
├── opengl     //OpenGL 相关
├── packages   //设置、TTS、VPN 程序
├── policy     //锁屏界面相关
├── sax        //XML 解析器
├── services   //Android 的服务
├── telephony  //电话相关
├── test-runner //测试相关
├── tests      //测试相关
├── tools      //工具
├── voip       //可视通话
└── wifi       //无线网络
```

以上这些文件夹包含了应用程序框架层的大部分代码，正是这些目录下的文件构成了 Android 的应用程序框架层，暴露出接口给应用程序调用，同时衔接系统程序库和硬件抽象层，形成一个由上至下的调用过程。

1.6.4　系统服务

Android 应用程序框架层的大部分实现代码被保存在/frameworks/base 目录下，其实在这个目录中还有一个名为 service 的目录，里面的代码用于实现 Android 系统服务，其目录结构如下所示：

```
frameworks/base/services
├── common_time        //日期时间相关的服务
├── input              //输入系统服务
├── java               //其他重要服务的 Java 层
├── jni                //其他重要服务的 JNI 层
└── tests              //测试相关
```

其中 java 和 jni 两个目录分别是一些其他的服务的 Java 层和 JNI 层实现，java 目录下的目录结构以及其他 Android 系统服务的相关说明如下所示：

```
frameworks/base/services/java/com/android/server
├── accessibility
├── am
├── connectivity
├── display
├── dreams
├── drm
├── input
├── location
├── net
├── pm
├── power
├── updates
├── usb
├── wm
├── AlarmManagerService.java            //闹钟服务
├── AppWidgetService.java               //应用程序小工具服务
├── AppWidgetServiceImpl.java
├── AttributeCache.java
├── BackupManagerService.java            //备份服务
├── BatteryService.java                 //电池相关服务
├── BluetoothManagerService.java         //蓝牙
├── BootReceiver.java
├── BrickReceiver.java
├── CertBlacklister.java
├── ClipboardService.java
├── CommonTimeManagementService.java     //时间管理服务
├── ConnectivityService.java
├── CountryDetectorService.java
├── DevicePolicyManagerService.java
├── DeviceStorageMonitorService.java     //设备存储器监听服务
├── DiskStatsService.java               //磁盘状态服务
```

```
├── DockObserver.java                   //底座监视服务
├── DropBoxManagerService.java
├── EntropyMixer.java
├── EventLogTags.logtags
├── INativeDaemonConnectorCallbacks.java
├── InputMethodManagerService.java      //输入法管理服务
├── IntentResolver.java
├── IntentResolverOld.java
├── LightsService.java
├── LocationManagerService.java         //地理位置服务
├── MasterClearReceiver.java
├── MountService.java                   //挂载服务
├── NativeDaemonConnector.java
├── NativeDaemonConnectorException.java
├── NativeDaemonEvent.java
├── NetworkManagementService.java       //网络管理服务
├── NetworkTimeUpdateService.java
├── NotificationManagerService.java     //通知服务
├── NsdService.java
├── PackageManagerBackupAgent.java
├── PreferredComponent.java
├── ProcessMap.java
├── RandomBlock.java
├── RecognitionManagerService.java
├── SamplingProfilerService.java
├── SerialService.java                  //NFC 相关
├── ServiceWatcher.java
├── ShutdownActivity.java
├── StatusBarManagerService.java        //状态栏管理服务
├── SystemBackupAgent.java
├── SystemServer.java
├── TelephonyRegistry.java
├── TextServicesManagerService.java
├── ThrottleService.java
├── TwilightCalculator.java
├── TwilightService.java
├── UiModeManagerService.java
├── UpdateLockService.java              //锁屏更新服务
├── VibratorService.java                //震动服务
├── WallpaperManagerService.java        //壁纸服务
├── Watchdog.java                       //看门狗
├── WifiService.java                    //无线网络服务
└── WiredAccessoryManager.java          //无线设备管理服务
```

从上面的文件夹和文件可以看出，Android 中涉及的服务种类有：界面、网络、电话等核心模块，这些专属服务是系统级别的服务，这些系统服务一般都会在 Android 系统启动的时候加载，在系统关闭的时候结束，受到系统的管理，应用程序并没有权力去打开或者关闭，它们会随着系统的运行一直在后台运行，供应用程序和其他组件来使用。

另外，在 frameworks/av/目录下面有一个 services 目录，在此目录中存放的是音频和照相机的服务的实现代码，此目录的具体结构如下所示：

```
frameworks/av/services
├── audioflinger       //音频管理服务
└── camera             //照相机的管理服务
```

av/services 目录主要用来支持 Android 系统中的音频和照相机服务。

1.6.5　系统程序库

Android 4.3 程序库的类型非常多，功能也非常强大。在接下来的内容中，将简要讲解 Android 4.3 源码中的一些常用并且重要的系统程序库的知识。

(1)　系统 C 库

Android 系统采用的是一个从 BSD 继承而来的标准的系统函数库 bionic，在源码根目录下有这个文件夹，其目录结构如下所示：

```
bionic/
├── libc            //C 库
├── libdl           //动态链接库相关
├── libm            //数学库
├── libstdc++       //C++实现库
├── libthread_db    //线程库
├── linker          //连接器相关
└── test            //测试相关
```

(2)　媒体库

Android 中的媒体库在 2.3 版之前是由 OpenCore 实现的，2.3 版之后 Stragefright 被替换了，OpenCore 成为新的多媒体的实现库。同时 Android 也自带了一些音视频的管理库，用于管理多媒体的录制、播放、编码和解码等功能。

Android 的多媒体程序库的实现代码主要在/frameworks/av/media 目录中，其目录结构如下：

```
frameworks/av/media/
├── common_time              //时间相关
├── libeffects               //多媒体效果
├── libmedia                 //多媒体录制、播放
├── libmedia_native          //里面只有一个 Android.mk，用来编译 native 文件
├── libmediaplayerservice    //多媒体播放服务的实现库
├── libstagefright           //Stagefright 的实现库
├── mediaserver              //跨进程多媒体服务
└── mtp                      //MTP 协议的实现(媒体传输协议)
```

(3)　图层显示库

Android 中的图层显示库主要负责对显示子系统的管理，负责图层的渲染、叠加、绘制等功能，提供了 2D 和 3D 图层的无缝融合，是整个 Android 系统显示的“大脑中枢”，其代码在/frameworks/native/services/surfaceflinger/目录下，其目录结构如下所示：

```
frameworks/native/services/surfaceflinger/
```

```
├─── DisplayHardware            //显示底层相关
├─── tests                      //测试
├─── Android.mk                 //MakeFile 文件
├─── Barrier.h
├─── Client.cpp                 //显示的客户端实现文件
├─── Client.h
├─── clz.cpp
├─── clz.h
├─── DdmConnection.cpp
├─── DdmConnection.h
├─── DisplayDevice.cpp          //显示设备相关
├─── DisplayDevice.h
├─── EventThread.cpp            //消息线程
├─── EventThread.h
├─── GLExtensions.cpp           //OpenGL 扩展
├─── GLExtensions.h
├─── Layer.cpp                  //图层相关
├─── Layer.h
├─── LayerBase.cpp              //图层基类
├─── LayerBase.h
├─── LayerDim.cpp               //图层相关
├─── LayerDim.h
├─── LayerScreenshot.cpp        //图层相关
├─── LayerScreenshot.h
├─── MessageQueue.cpp           //消息队列
├─── GLExtensions.h
├─── MessageQueue.h
├─── MODULE_LICENSE_APACHE2     //证书
├─── SurfaceFlinger.cpp         //图层管理者，图层管理的核心类
├─── SurfaceFlinger.h
├─── SurfaceTextureLayer.cpp    //文字图层
├─── SurfaceTextureLayer.h
├─── Transform.cpp
└─── Transform.h
```

(4) 网络引擎库

网络引擎库主要是用来实现 Web 浏览器的引擎，支持 Android 的 Web 浏览器和一个可嵌入的 Web 视图，这是采用第三方开发的浏览器引擎 Webkit 实现的，Webkit 的代码在/external/webkit/目录下，其目录结构如下所示：

```
external/webkit/
├─── Examples               //Webkit 的例子
├─── LayoutTests            //布局测试
├─── PerformanceTests       //表现测试
├─── Source                 //Webkit 源代码
├─── Tools                  //工具
├─── WebKitLibraries        //Webkit 用到的库
├─── Android.mk             //Makefile
├─── bison_check.mk
```

```
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL     //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION   //版本信息
```

(5)　3D 图形库

Android 中的 3D 图形渲染是采用 OpenGL 来实现的，OpenGL 是开源的第三方图形渲染库，使用该库可以实现 Android 中的 3D 图形硬件加速或者 3D 图形软件加速功能，是一个非常重要的功能库。从 Android 4.3 开始，支持最新、最强大的 OpenGL ES 3.0。其实现代码在/frameworks/native/opengl 中，其目录结构如下所示：

```
frameworks/native/opengl/
├── include                //OpenGL 中的头文件
├── libagl                 //在 Mac OS 上的库
├── libs                   //OpenGL 的接口和实现库
├── specs                  //OpenGL 的文档
├── tests                  //测试相关
└── tools                  //工具库
```

(6)　SQLite

SQLite 是 Android 系统自带的一个轻量级关系数据库，其实现源代码已经在网上开源。SQLite 的优点是操作简单方便，运行速度较快，占用资源较少等，比较适合在嵌入式设备上面使用。SQLite 是 Android 系统自带的实现数据库功能的核心库，其代码实现分为 Java 和 C 两个部分，Java 部分的代码位于/frameworks/base/core/java/android/database，目录结构如下所示：

```
frameworks/base/core/java/android/database/
├── sqlite                                   //SQLite 的框架文件
├── AbstractCursor.java                      //游标的抽象类
├── AbstractWindowedCursor.java
├── BulkCursorDescriptor.java
├── BulkCursorNative.java
├── BulkCursorToCursorAdaptor.java           //游标适配器
├── CharArrayBuffer.java
├── ContentObservable.java
├── ContentObserver.java                     //内容观察者
├── CrossProcessCursor.java
├── CrossProcessCursorWrapper.java           //CrossProcessCursor 的封装类
├── Cursor.java                              //游标实现类
├── CursorIndexOutOfBoundsException.java     //游标出界异常
├── CursorJoiner.java
├── CursorToBulkCursorAdaptor.java           //适配器
├── CursorWindow.java                        //游标窗口
├── CursorWindowAllocationException.java     //游标窗口异常
├── CursorWrapper.java                       //游标封装类
├── DatabaseErrorHandler.java                //数据库错误句柄
├── DatabaseUtils.java                       //数据库工具类
├── DataSetObservable.java
├── DataSetObserver.java
```

```
├── DefaultDatabaseErrorHandler.java        //默认数据库错误句柄
├── IBulkCursor.java
├── IContentObserver.aidl                   //aidl 用于跨进程通信
├── MatrixCursor.java
├── MergeCursor.java
├── Observable.java
├── package.html
├── SQLException.java                       //数据库异常
└── StaleDataException.java
```

Java 层的代码主要是实现 SQLite 的框架和接口的实现，使用户开发应用程序时能很简单地操作数据库，并且捕获数据库异常。

C++层的代码在/external/sqlite 路径下，其目录结构如下所示：

```
external/sqlite/
├── android        //Android 数据库的一些工具包
└── dist           //Android 数据库底层实现
```

从上面 Java 和 C 部分的代码目录结构可以看出，SQLite 在 Android 中还是有很重要的地位的，并且在 SDK 中会有开放的接口让应用程序可以很简单方便地操作数据库，对数据进行存储和删除。

1.6.6 系统运行库

众所周知，Android 系统的应用层是采用 Java 开发的，由于 Java 语言的跨平台特性，Java 代码必须运行在虚拟机中。正是因为这个特性，Android 系统也自己实现了一个类似 JVM 但是更适用于嵌入式平台的 Java 虚拟机，这被称为 dalvik。

dalvik 功能等同于 JVM，为 Android 平台上的 Java 代码提供了运行环境，dalvik 本身是由 C++语言实现的，在源码中的根目录下有 dalvik 文件夹，里面存放的是 dalvik 虚拟机的实现代码，其目录结构如下所示：

```
./
├── dalvikvm                //入口目录
├── dexdump                 //dex 反汇编
├── dexgen                  //dex 生成相关
├── dexlist                 //dex 列表
├── dexopt                  //与验证和优化
├── docs                    //文档
├── dvz                     //zygot 相关
├── dx                      //dx 工具，将多个 Java 转换为 dex
├── hit
├── libdex                  //dex 库的实现代码
├── opcode-gen
├── tests                   //测试相关
├── tools                   //工具
├── unit-tests              //测试相关
├── vm                      //虚拟机的实现
├── Android.mk              //Makefile
```

```
├── CleanSpec.mk
├── MODULE_LICENSE_APACHE2
├── NOTICE
└── README.txt
```

正是因为有上面这些代码实现的 Android 虚拟机，所以应用程序生成的二进制执行文件能够快速、稳定地运行在 Android 系统上。

1.6.7　硬件抽象层

Android 的硬件抽象是各种功能的底层实现，理论上，不同的硬件平台会有不同的硬件抽象层实现，这一个层次也是与驱动层和硬件层有紧密联系的，起着承上启下的作用，对上要实现应用程序框架层的接口，对下要实现一些硬件的基本功能，以及调用驱动层的接口。需要注意的是，这一层也是广大 OEM 厂商改动最大的一层，因为这一层的代码跟终端采用什么样硬件的硬件平台有很大的关系。源码中存放的是硬件抽象层框架的实现代码和一些平台无关的接口的实现。硬件抽象层代码在源码根目录下的 hardware 文件夹中，其目录结构如下所示：

```
hardware/
├── libhardware              //新机制硬件库
├── libhardware_legacy       //旧机制硬件库
└── ril                      //ril 模块相关的底层实现
```

从上面的目录结构我们可以看出，硬件抽象层中主要是实现了一些底层的硬件库，用来实现应用层框架中的功能，至于具体硬件库中有哪些内容，我们可以继续细分其目录结构，例如 libhardware 目录下的结构为：

```
hardware/libhardware/
├── include                      //入口目录
├── modules                      //dex 反汇编
│    ├── audio                   //音频相关底层库
│    ├── audio_remote_submix     //音频混合相关
│    ├── gralloc                 //帧缓冲
│    ├── hwcomposer              //音频相关
│    ├── local_time              //本地时间
│    ├── nfc                     //nfc 功能
│    ├── nfc-nci                 //nfc 的接口
│    ├── power                   //电源
│    ├── usbaudio                //USB 音频设备
│    ├── Android.mk              //Makefile
│    ├── README.android
├── tests                        //dex 生成相关
├── dexlist                      //dex 列表
├── dexopt                       //验证和优化
└── docs                         //文档
```

从上面的目录结构我们可以分析出，libhardware 目录主要是 Android 系统的某些功能的底层实现，包括 audio、nfc、power。

libhardware_legacy 目录与 libhardware 大同小异，只是针对旧的实现方式做的一套硬件库，

其目录下还有 uevent、wifi 以及虚拟机的底层实现。这两个目录下的代码一般会由设备厂家根据自身的硬件平台来实现符合 Android 机制的硬件库。

ril 目录下存放的是无线硬件设备与电话的实现，其目录结构如下所示：

```
hardware/ril/
├── include              //头文件
├── libril               //libril 库
├── mock-ril
├── reference-ril         //reference ril 库
├── rild                 //ril 守护进程
└── CleanSpec.mk
```

1.7 编译Android源码

编译 Android 源码的方法非常简单，只需使用 Android 源码根目录下的 Makefile，执行 make 命令即可轻松实现。当然，在编译 Android 源码之前，首先要确定已经完成了同步工作。进入 Android 源码目录，使用 make 命令进行编译，使用此命令的格式如下所示：

```
$: cd ~/Android4.3(这里的 Android4.3 就是我们下载源码的保存目录)
$: make
```

编译 Android 源码可以得到~/project/android/cupcake/out 目录，作者计算机上的界面截图如图 1-14 所示。

```
主意: external/doclava/src/com/google/doclava/Stubs.java 使用了未经检查或不安全
的操作。
主意: 要了解详细信息, 请使用 -Xlint:unchecked 重新编译。
Notice file: external/doclava/NOTICE -- out/host/linux-x86/obj/NOTICE_FILES/src/
/framework/doclava.jar.txt
Install: out/host/linux-x86/framework/doclava.jar
target Java: core (out/target/common/obj/JAVA_LIBRARIES/core_intermediates/class
es)
主意: 某些输入文件使用或覆盖了已过时的 API。
主意: 要了解详细信息, 请使用 -Xlint:deprecation 重新编译。
主意: 某些输入文件使用了未经检查或不安全的操作。
主意: 要了解详细信息, 请使用 -Xlint:unchecked 重新编译。
Copying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes-jarjar.
jar
Copying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/emma_out/lib/cl
asses-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes.jar
target Java: conscrypt (out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermedi
ates/classes)
主意: 某些输入文件使用或覆盖了已过时的 API。
主意: 要了解详细信息, 请使用 -Xlint:deprecation 重新编译。
主意: 某些输入文件使用了未经检查或不安全的操作。
主意: 要了解详细信息, 请使用 -Xlint:unchecked 重新编译。
host Prebuilt: jarjar (out/host/common/obj/JAVA_LIBRARIES/jarjar_intermediates/javalib.jar)
Install: out/host/linux-x86/framework/jarjar.jar
JarJar: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/emma_out/lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/classes.jar
target Java: bouncycastle (out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes)
主意: 某些输入文件使用或覆盖了已过时的 API。
主意: 要了解详细信息, 请使用 -Xlint:deprecation 重新编译。
主意: 某些输入文件使用了未经检查或不安全的操作。
主意: 要了解详细信息, 请使用 -Xlint:unchecked 重新编译。
JarJar: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/emma_out/lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes.jar
target Java: ext (out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/classes)
```

图 1-14 编译过程的界面截图

整个编译过程也是非常长的，需要耐心等待。本节的内容中，将详细讲解编译 Android 4.3

源码的方法。

1.7.1　搭建编译环境

在编译 Android 源码之前，需要先进行环境搭建工作。在接下来的内容中，以 Ubuntu 系统为例，讲解搭建编译环境以及编译 Android 源码的方法。具体流程如下所示。

(1)　安装 JDK，编译 Android 4.3 的源码需要 JDK 1.6，下载 jdk-6u21-linux-i586.bin 后进行安装，对应的命令如下：

```
$ cd /usr
$ mkdir java
$ cd java
$ sudo cp jdk-6u21-linux-i586.bin 所在目录 ./
$ sudo chmod 755 jdk-6u21-linux-i586.bin
$ sudo sh jdk-6u21-linux-i586.bin
```

(2)　设置 JDK 环境变量，将如下环境变量添加到主文件夹目录下的.bashrc 文件中，然后用 source 命令使其生效，加入的环境变量代码如下：

```
export JAVA_HOME=/usr/java/jdk1.7.0_1
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/bin/tools.jar:$JRE_HOME/bin
export ANDROID_JAVA_HOME=$JAVA_HOME
```

(3)　安装需要的包，读者可以根据编译过程中的提示进行选择，可能需要的包的安装命令如下：

```
$ sudo apt-get install git-core bison zlib1g-dev flex libx11-dev gperf sudo aptitude
install git-core gnupg flex bison gperf libsdl-dev libesd0-dev libwxgtk2.6-dev
build-essential zip curl libncurses5-dev zlib1g-dev
```

1.7.2　开始编译

当所依赖的包安装完成之后，就可以开始编译 Android 源码了，具体步骤如下所示。

(1)　首先进行编译初始化工作，在终端中执行下面的命令：

```
source build/envsetup.sh
```

或者：

```
. build/envsetup.sh
```

执行后将会输出：

```
source build/envsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/armv7-a/vendorsetup.sh
```

```
including device/generic/mips/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including device/samsung/toroplus/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

(2) 然后选择编译目标，命令如下：

```
lunch full-eng
```

执行后，会输出如下所示的提示信息：

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-2.6.31-45-generic-x86_64-with-Ubuntu-10.04-lucid
HOST_BUILD_TYPE=release
BUILD_ID=JOP40C
OUT_DIR=out
============================================
```

(3) 接下来开始编译代码，在终端中执行下面的命令：

```
make -j4
```

其中，-j4 表示用 4 个线程进行编译。整个编译进度根据不同机器的配置而需要不同的时间。例如，作者所用的电脑为 Intel i7 处理器，四核 2.4GHz，4GB 的内存，经过近 4 小时才编译完成。当出现如下所示的提示信息时，表示完成了编译工作：

```
target Java: ContactsTests
(out/target/common/obj/APPS/ContactsTests_intermediates/classes)
target Dex: Contacts
Done!
Install: out/target/product/generic/system/app/Browser.odex
Install: out/target/product/generic/system/app/Browser.apk
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Copying:
out/target/common/obj/APPS/Contacts_intermediates/noproguard.classes.dex
target Package: Contacts
(out/target/product/generic/obj/APPS/Contacts_intermediates/package.apk)
```

```
'out/target/common/obj/APPS/Contacts_intermediates/classes.dex' as
'classes.dex'...
Processing target/product/generic/obj/APPS/Contacts_intermediates/package.apk
Done!
Install: out/target/product/generic/system/app/Contacts.odex
Install: out/target/product/generic/system/app/Contacts.apk
build/tools/generate-notice-files.py
out/target/product/generic/obj/NOTICE.txt
out/target/product/generic/obj/NOTICE.html "Notices for files contained in the
filesystem images in this directory:"
out/target/product/generic/obj/NOTICE_FILES/src
Combining NOTICE files into HTML
Combining NOTICE files into text
Installed file list: out/target/product/generic/installed-files.txt
Target system fs image:
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
Running:  mkyaffs2image -f out/target/product/generic/system
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
Install system fs image: out/target/product/generic/system.img
DroidDoc took 5331 sec. to write docs to out/target/common/docs/doc-comment-check
```

1.7.3　在模拟器中运行

在模拟器中运行编译源码的步骤比较简单，只需在终端中执行下面的命令即可：

```
emulator
```

运行成功后的效果如图 1-15 所示。

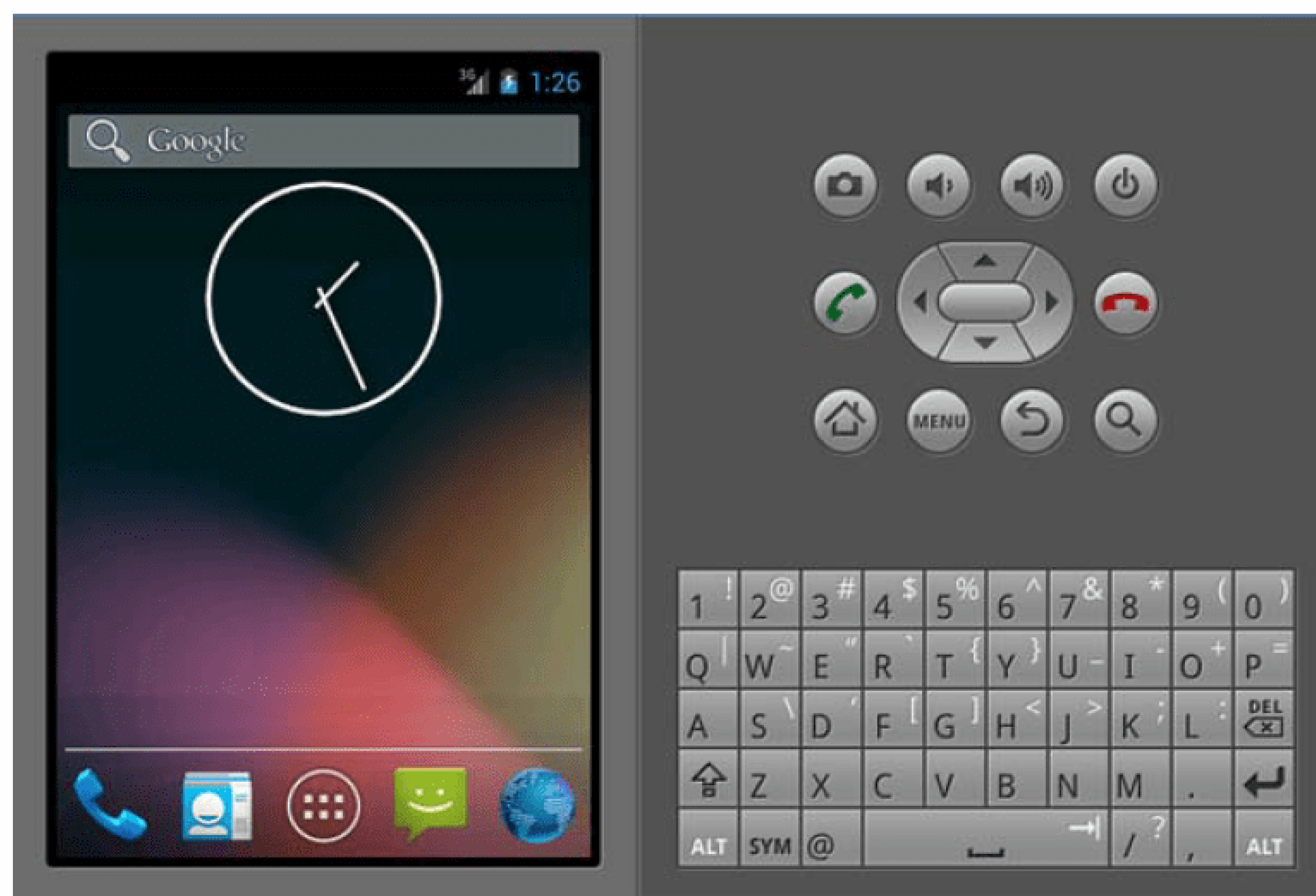


图 1-15　在模拟器中的编译执行效果

1.7.4　编译源码生成SDK

在现实应用中，通常是基于 SDK 来实现 Android 应用程序开发工作的，其过程是使用 SDK 中的接口来实现各种各样的功能。最常规的做法是在 Android 的官方网站上面直接下载最新的 SDK 版本，然后搭建应用开发环境。其实除此之外，我们也可以从源码中生成 SDK，因为源码里面也包含有 SDK 的代码。

在下载的 Android 4.3 的源码的根目录中有一个 SDK 目录，所有的 SDK 相关的代码都放在这个目录中，包括镜像文件、模拟器、ADB 等常用工具以及 SDK 中的开发包的文档。可以通过编译的方式来生成开发需要的 SDK，具体的编译命令如下：

```
$ Make SDK
```

编译完成后，会在/out/host/linux-x86/sdk/目录下生成 SDK，这个 SDK 是完全跟源码同步的，与官网上下载的 SDK 功能完全相同，会有开发用的 JAR 包、模拟器管理工具、ADB 调试工具，可以使用这个编译生成的 SDK 来开发我们的应用程序。

对于 Android 系统的开发，基本可以分为如下两种开发方式：

- 基于 SDK 的开发。
- 基于源码的开发。

在一般情况下，开发的应用程序都是基于 SDK 的开发，比较方便而且兼容性比较好。基于源码的开发相对于基于 SDK 的开发要求对源码的架构认识更深刻，一般用于需要修改系统层面的场合。两种方式应用场景不同，各有优缺点，本节将主要介绍基于 SDK 的开发。

如果想基于 SDK 开发 Android 的应用程序，我们需要 JDK、SDK 和一个开发环境，JDK 和 SDK 在不同的平台下有不同的版本，本章主要讨论 Windows 7 平台下的开发环境搭建。

(1)　安装 JDK

由于 Android 的应用程序使用 Java 语言开发，所以首先需要安装 Java 的 JDK，下载链接是 http://java.sun.com/javase/downloads/index.jsp，进入页面后，选择合适的平台以及下载最新版本的 JDK，安装成功后，可以在命令行下查看 JDK 版本，如图 1-16 所示。

```
C:\Windows\system32>java -version
java version "1.6.0_25"
Java(TM) SE Runtime Environment (build 1.6.0_25-b06)
Java HotSpot(TM) Client VM (build 20.0-b11, mixed mode, sharing)
```

图 1-16　成功安装JDK

(2)　安装 Eclipse

Eclipse 是开发 Android 应用程序的 IDE 环境，有非常丰富的插件可以使用，通过下载网址 http://www.eclipse.org/downloads/可以下载合适平台的最新版本的 Eclipse。

(3)　安装 Android SDK

Android SDK 是 Google 对外发布的专门用于 Android 开发的工具包，里面有各种版本的开发框架和工具，以及丰富的文档，打开 http://developer.android.com/sdk/index.html 页面，可以下载最新版本的针对 Window 7 平台的 SDK。

当下载完成上述三个工具之后，需要对开发环境进行如下所示的配置。

1．配置Eclipse

(1) 打开 Eclipse，选择 help → Install New Software 菜单命令，出现图 1-17 给出的界面。

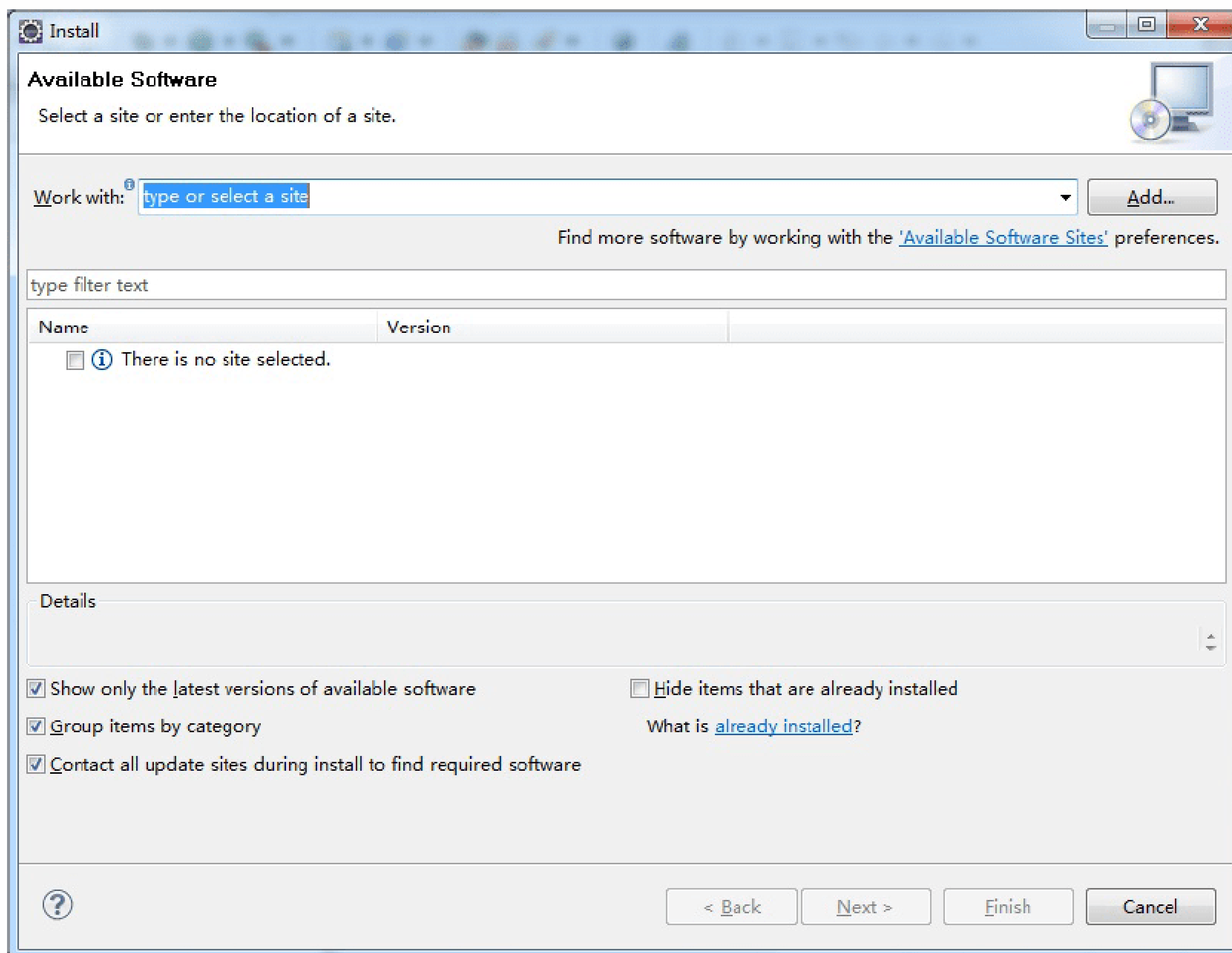


图 1-17　Available Software界面

(2) 单击 Add 按钮，会出现如图 1-18 所示的对话框。

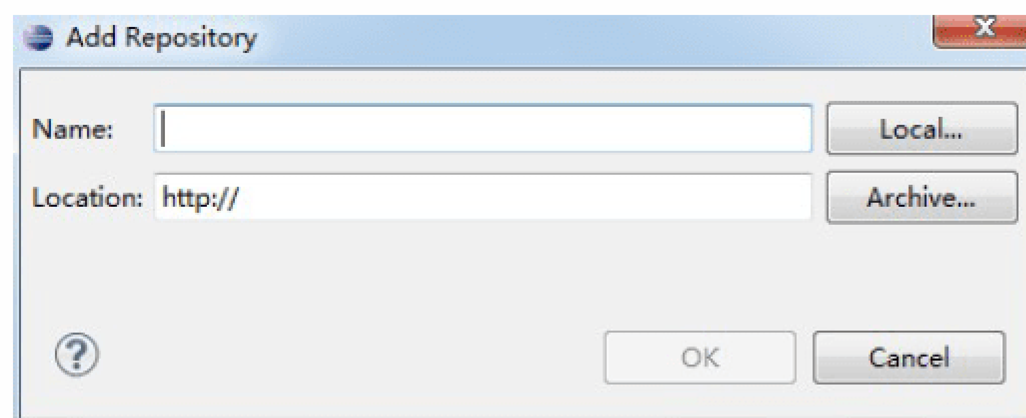


图 1-18　Add Repository对话框

(3) 在 Name 文本框中输入“Android”或者自定义任何名字，然后在 Location 文本框中键入“https://dl-ssl.google.com/android/eclipse/”，键入后的效果如图 1-19 所示。

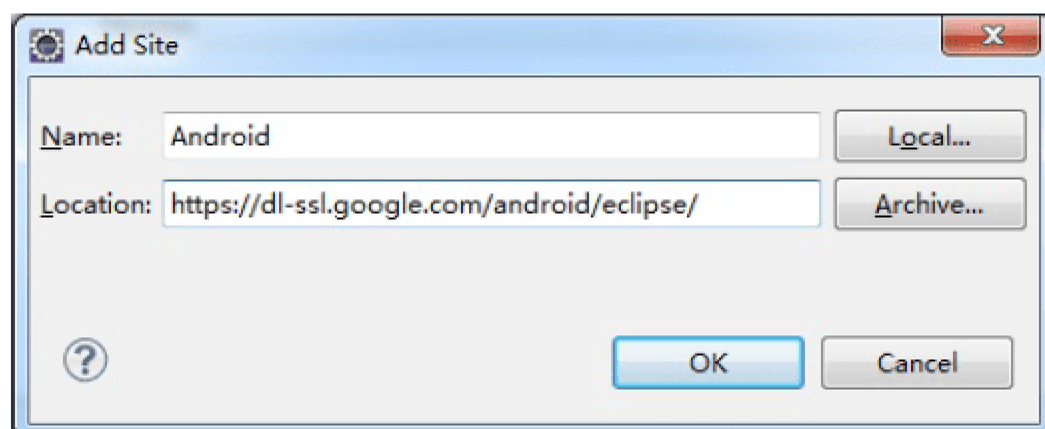


图 1-19　键入后的效果

(4) 如果发现 https://无法使用，可以改成 http://尝试一下，当输入好名字和地址之后，单击 OK 按钮，会出现如图 1-20 所示的界面。

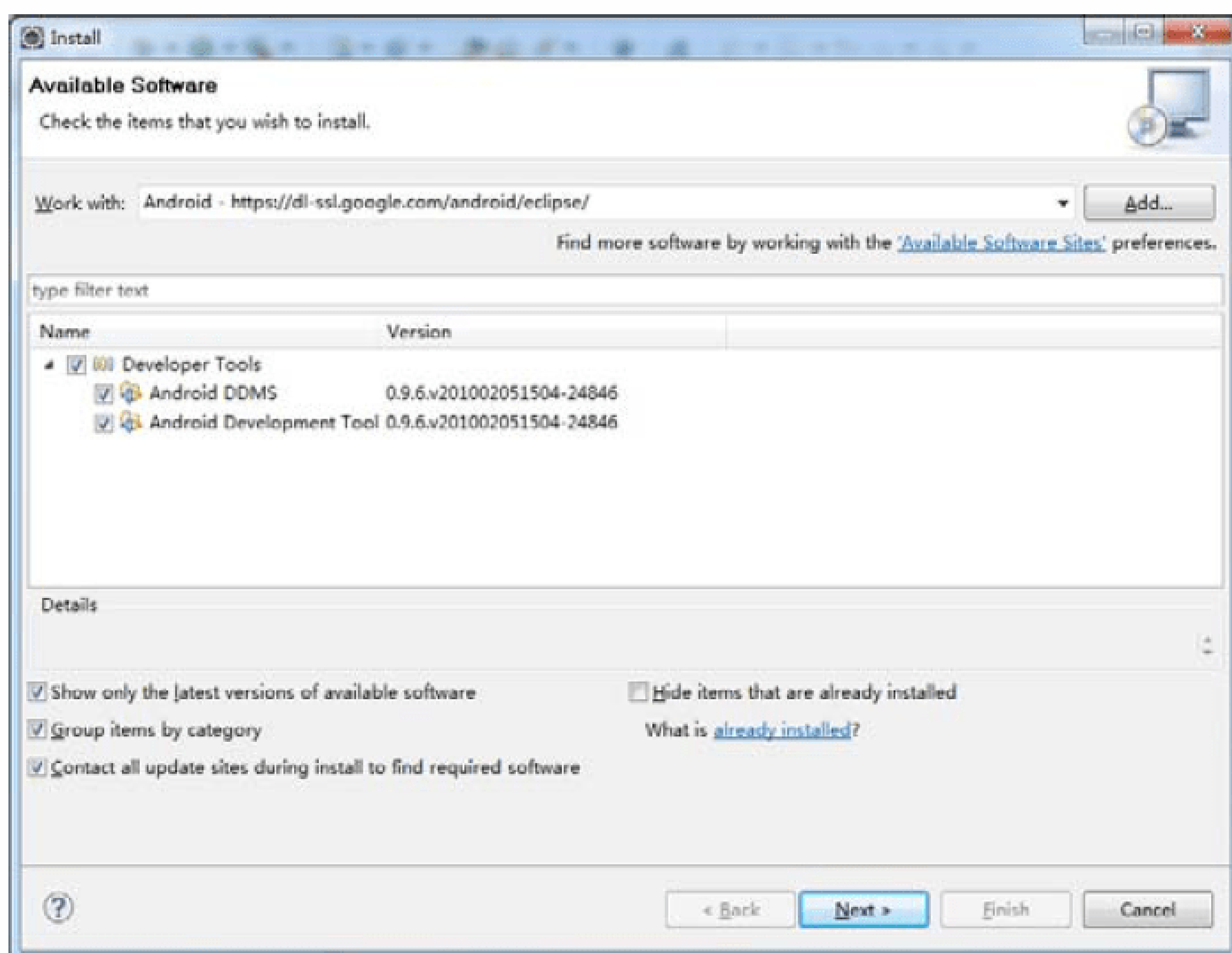


图 1-20 输入名字和地址后的界面

图 1-20 中的两个插件都是 Android 开发必不可少的工具包，Android DDMS 可以用来调试，管理 Android 进程、存储器，查看日志，Android Development Tool 简称 ADT，是开发 Android 的插件，只有装了 ADT，才能创建 Android 工程。

(5) 单击 Next 按钮，出现如图 1-21 所示的界面。

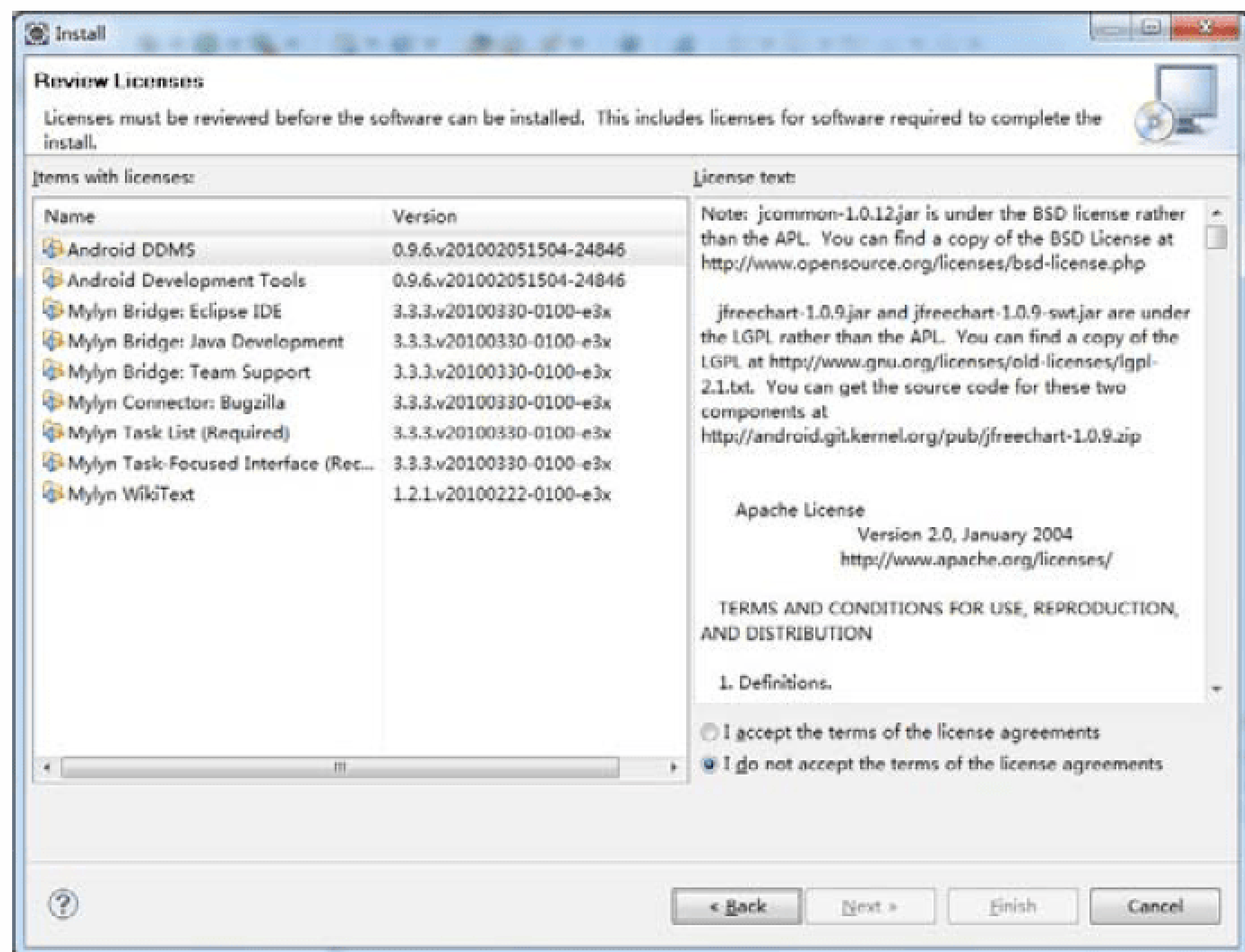


图 1-21 Review Licenses界面

在图 1-21 中列出了将会安装的工具包。选中“I accept…”选项，单击 Next 按钮，会开始安装插件，界面如图 1-22 所示。

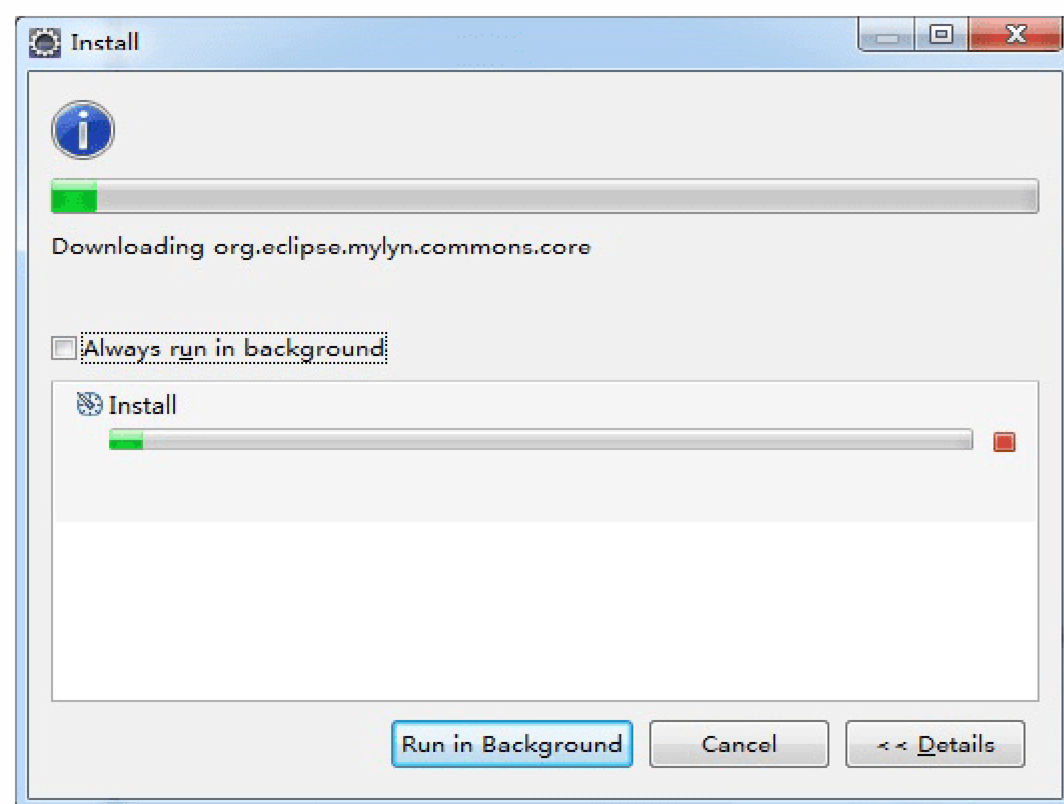


图 1-22　开始安装

(6)　当所有插件安装成功后，会弹出提示对话框，如图 1-23 所示。

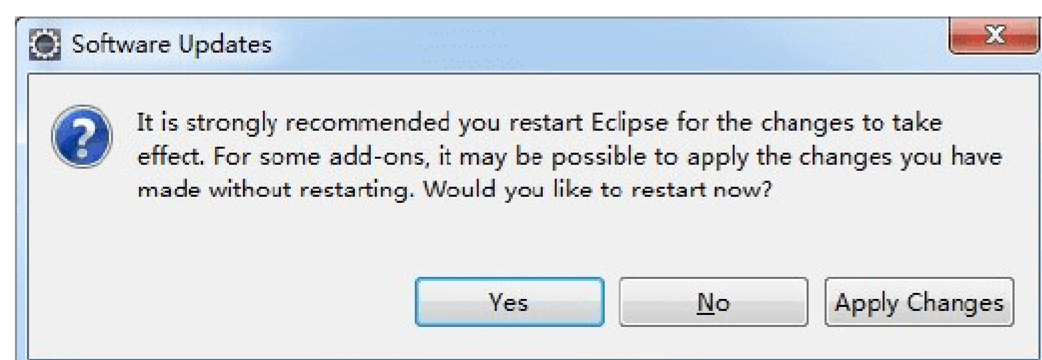


图 1-23　提示对话框

这时我们需要单击 Yes 按钮重启 Eclipse，让所有插件生效。

2．配置Android SDK

打开 Eclipse，选择 Window → Preferences 命令，出现如图 1-24 所示的界面，按照该图进行配置即可。

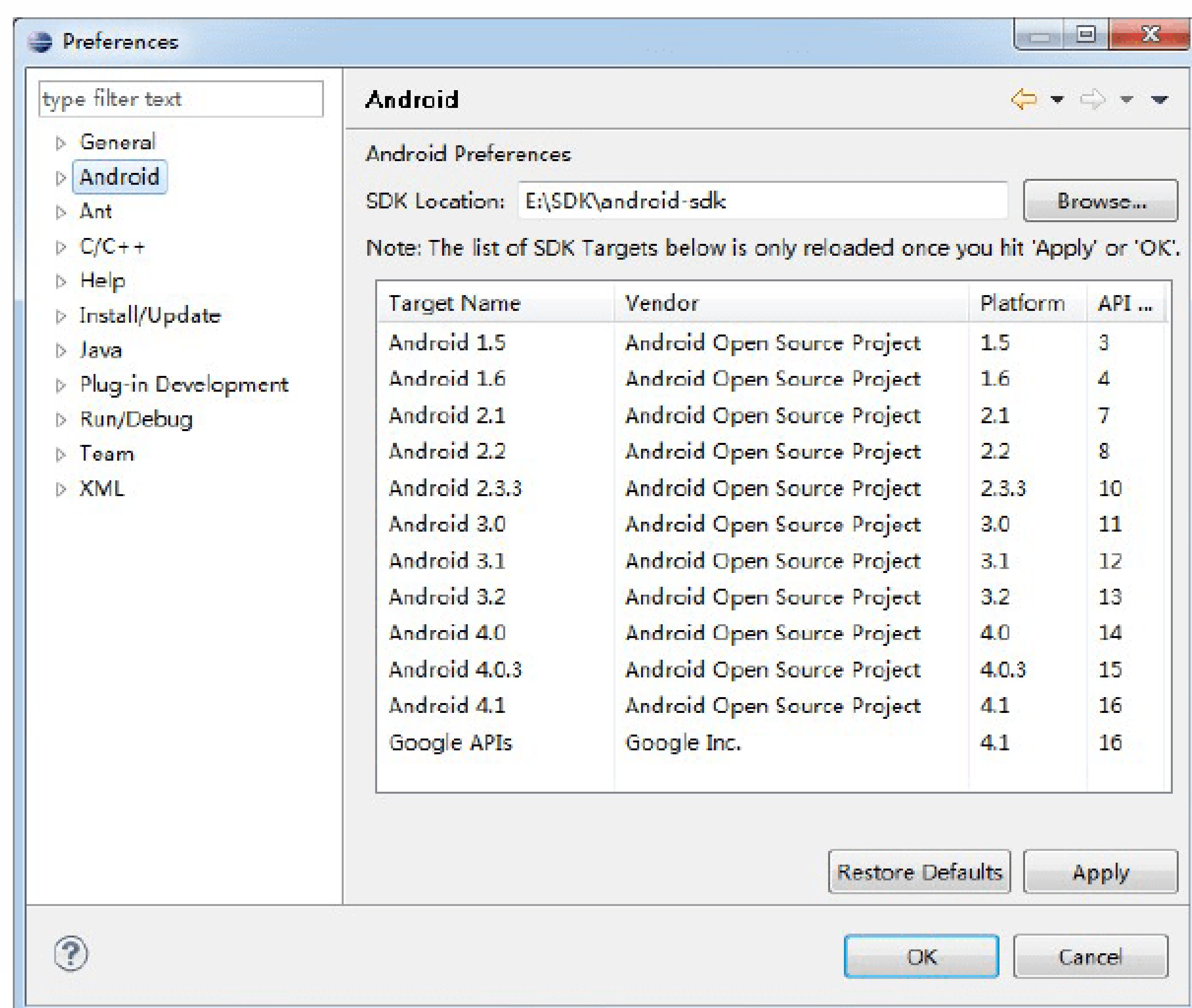


图 1-24　配置界面

这样，我们就可以从 Eclipse 中新建 Android 工程了。要想新建工程，应该明确是基于什么版本的 Android 系统，可以打开 SDK 根目录下的 SDK 管理工具 SDK Manager.exe，双击后，会进入到 SDK 工具包管理界面，如图 1-25 所示。

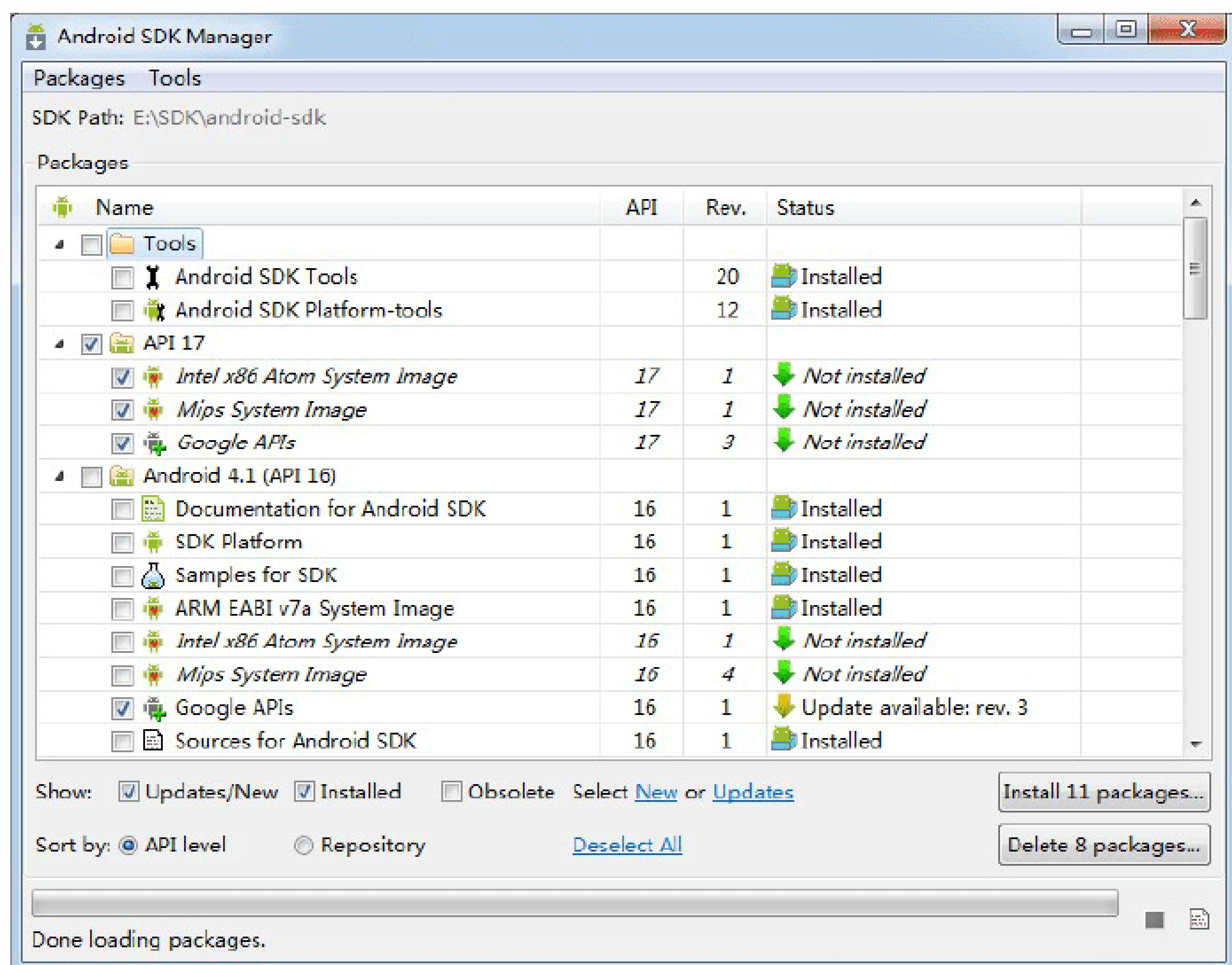


图 1-25　Android SDK管理

在图 1-25 中可以看到，这里很清晰地列出了当前版本 SDK 中包含的工具包，以及已经安装了的和没有安装的版本。可以继续单击 Install xx Packages 或者 Delete xx Packages 按钮安装或删除 SDK 中的工具包。如果是安装，则过程会比较慢，这与网速的关系比较大。

我们将 SDK 中的工具包安装完毕，同时也完成了 Eclipse 和 SDK 的配置工作后，Windows 7 平台下基于 SDK 的 Android 的开发环境就全部搭建完成了。

Android

第 2 章
硬件抽象层详解

在 Android 系统中，硬件抽象层(Hardware Abstract Layer，HAL)在用户空间中运行，向下能够屏蔽硬件驱动模块的实现细节，向上能够提供硬件访问服务。

Android 系统通过硬件抽象层，分两层来支持硬件设备，其中一层在用户空间中实现，一层在内核空间中实现。

在本章的内容中，将详细讲解 Android 4.3 系统中的硬件抽象层的基本知识，为读者步入本书后面高级知识的学习打下基础。

2.1 什么是HAL层

HAL 层是位于操作系统内核与硬件电路之间的接口层，其功能是将硬件抽象化。HAL 层隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，这样就可以在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，从而使软硬件测试工作的并行进行成为可能。

2.1.1 为什么把对硬件的支持划分为两层来实现

Android 系统为什么要把对硬件的支持划分为两层来实现呢？具体原因如下所示。

(1) Linux 内核代码遵循 GPL1 协议，如果在 Android 系统所使用的 Linux 内核中添加或者修改了代码，那么就必须将它们公开。因此，如果 Android 系统像其他的 Linux 系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能会损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。

(2) Android 系统代码是遵循 Apache License 2 协议的，它允许移动设备厂商添加或者修改 Android 系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在 Android 系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，它封装了硬件的实现细节和参数。这样就可以保护移动设备厂商的利益了。

2.1.2 HAL层的位置结构

HAL 层的位置结构如图 2-1 所示。



图 2-1 HAL层的位置结构

从图 2-1 可以看出，HAL 的功能是把 Android 框架与 Linux 内核隔离。这样做的目的是让 Android 不过度依赖 Linux 内核，从而让 Android 框架开发可以在不考虑驱动程序的前提下进行。在 HAL 层，主要包含了 GPS、Vibrator、Wi-Fi、Copybit、Audio、Camera、Lights、Ril、

Overlay 等模块。

(1) 硬件抽象层划分。

在 Android 系统中，硬件抽象层可以分为如下 6 种 HAL：

- 上层软件。
- 内部以太网。
- 内部通信 Client。
- 用户接入口。
- 虚拟驱动，设置管理模块。
- 内部通信 Server。

(2) 硬件抽象层接口的代码的特点。

定义硬件抽象层接口的代码具有以下 5 个特点：

- 硬件抽象层具有与硬件的密切相关性。
- 硬件抽象层具有与操作系统的无关性。
- 接口定义的功能应包含硬件或系统所需硬件支持的所有功能。
- 接口定义简单明了，太多的接口函数会增加软件模拟的复杂性。
- 具有可测性的接口设计有利于系统的软硬件测试和集成。

(3) HAL 的保存位置。

在 Android 源码中，HAL 主要被保存在如下所示的目录中。

- libhardware_legacy：过去的目录，采取了链接库模块观念来架构。
- libhardware：新版的目录，被调整为用 HAL stub 观念来架构。
- ril：是 Radio 接口层。
- msm7k：与 QUAL 平台相关的信息。

到目前为止，Android 的 HAL 层仍旧分布在不同的地方，例如分为 Camera、Wi-Fi 等，因此上述目录并不包含所有的 HAL 程序代码。在 HAL 架构成熟前的结构如图 2-2 所示，现在的 HAL 层的结构如图 2-3 所示。

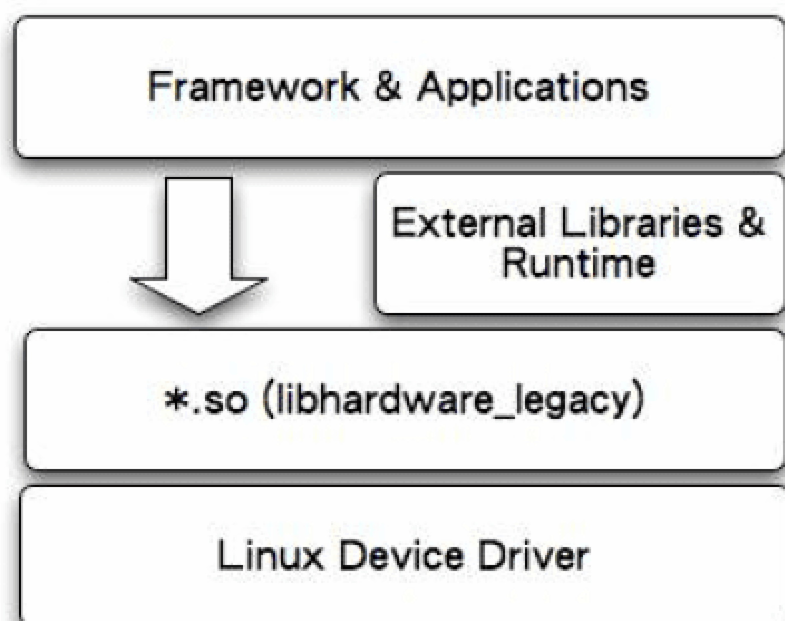


图 2-2 成熟前的HAL架构

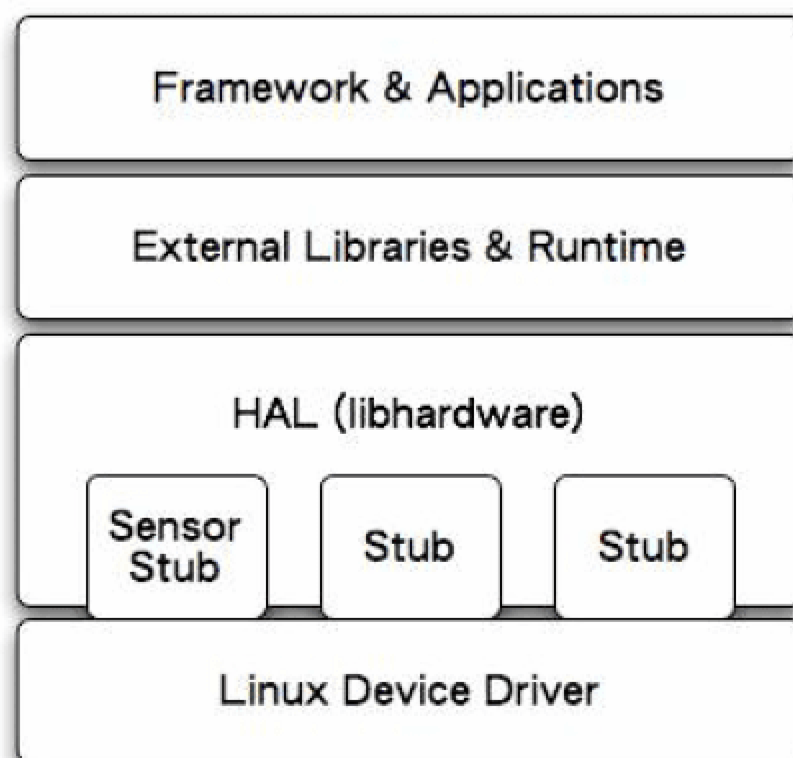


图 2-3 现在的HAL架构

从图 2-2 和 2-3 中的 HAL 层结构可以看出，当前的 HAL Stub 模式是一种代理人(proxy)的概念，虽然 Stub 仍以“*.so”文件的形式存在，但是 HAL 已经将“*.so”文件隐藏了。Stub

向 HAL 提供了功能强大的操作函数(Operations)，而 Runtime 则从 HAL 获取特定模块(Stub)的函数，然后再回调这些操作函数。这种“间接函数调用”模式的架构，让 HAL Stub 变成了一种“包含”关系，也就说，在 HAL 里包含了许多 Stub(代理人)。Runtime 只要说明模块 ID(类型)就可以取得操作函数。在当前的 HAL 模式中，Android 定义了 HAL 层结构框架，这样，通过接口访问硬件时，就形成了统一的调用方式。

2.2 分析HAL Module架构

在 Android 系统的源码中，在 Linux 内核和用户空间之间提供了一个 HAL 层，即硬件抽象层。这使得 Android 中的 Framework 只需要关心 HAL 中的内容，而不用关心具体的硬件实现。现在的 HAL 系统采用 HAL Module 和 HAL Stub 结合的形式，HAL Stub 不是一个共享库，编译时，上层只拥有访问 HAL Stub 的函数指针，并不需要 HAL Stub。上层通过 HAL Module 提供的统一接口获取并操作 HAL Stub，*.so 文件只会被映射到一个进程，也不存在重复映射和重入问题。

在 HAL Module 中，主要分为如下三个结构体：

- struct hw_module_t
- struct hw_module_methods_t
- struct hw_device_t

上述三个结构体的继承关系如图 2-4 所示。

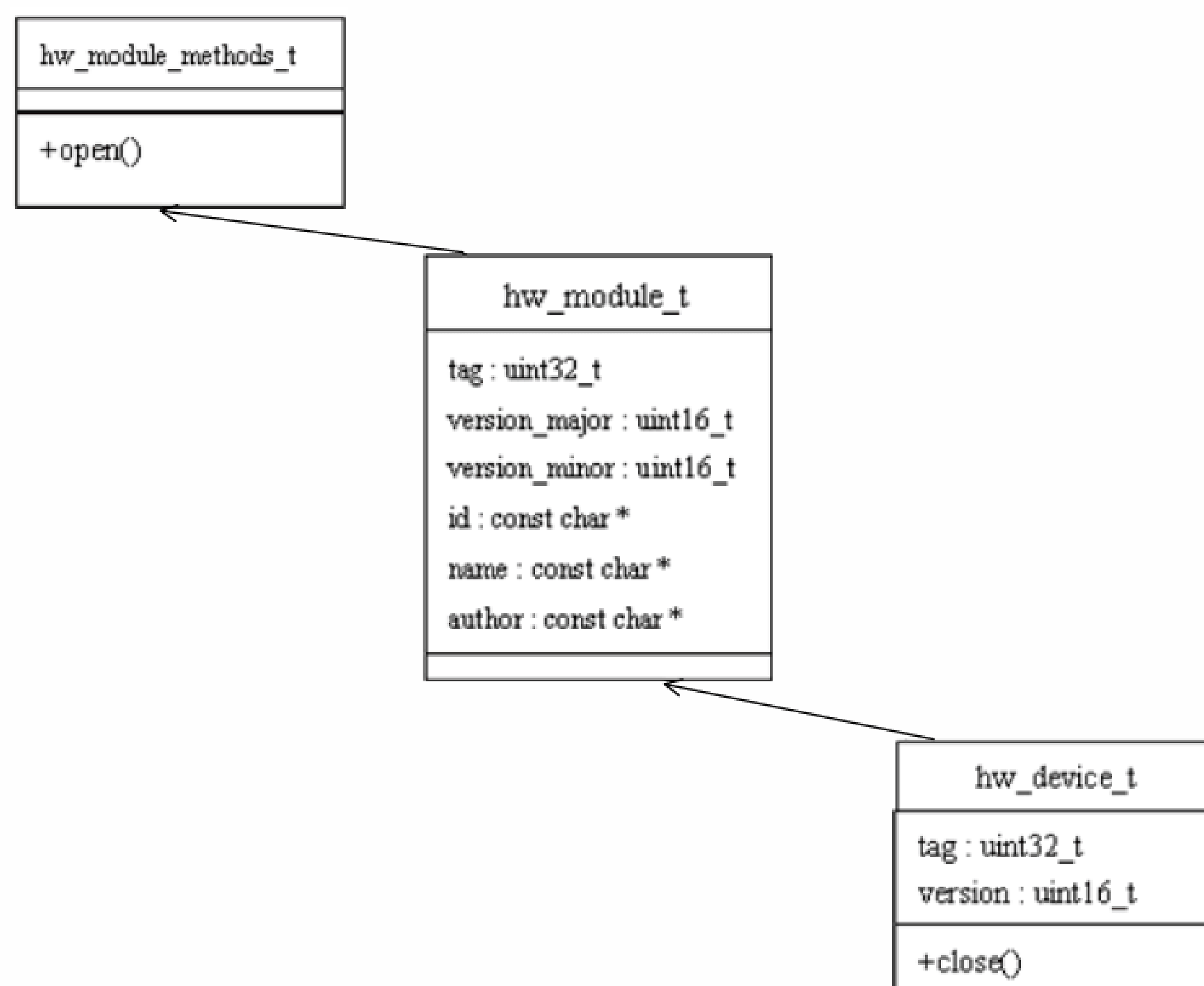


图 2-4　Android HAL结构体的继承关系

以上三个抽象概念在文件 hardware.c 中进行了具体描述，而 HAL 模块的源代码保存在 hardware 目录中。对于不同的 hardware 的 HAL，对应的 lib 命名规则是“id.variant.so”，比如

gralloc.msm7k.so 表示其 id 是 gralloc，msm7k 是 variant(变量)。variant 的取值范围是在该文件中定义的 variant_keys 对应的值。

2.2.1　hw_module_t

结构体 hw_module_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下：

```
typedef struct hw_module_t {
    uint32_t tag;
    uint16_t module_api_version;
#define version_major module_api_version
    uint16_t hal_api_version;
#define version_minor hal_api_version
    const char *id;
    const char *name;
    const char *author;
    struct hw_module_methods_t *methods;
    void *dso;
    uint32_t reserved[32-7];
} hw_module_t;
```

在结构体 hw_module_t 中，读者需要注意如下 5 点。

(1)　在结构体 hw_module_t 的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为 hw_module_t。

(2)　硬件抽象层中的每一个模块都必须存在一个导出符号 HAL_MODULE_IFNO_SYM，即 HMI，它指向一个自定义的硬件抽象层模块结构体。后面我们在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

(3)　结构体 hw_module_t 的成员变量 tag 的值必须设置为 HARDWARE_MODULE_TAG，即设置为一个常量值(‘H’<<24 | ‘W’<<16 | ‘M’<<8 | ‘T’)，标识这是一个硬件抽象层模块结构体。

(4)　结构体 hw_module_t 的成员变量 dso 用来保存加载硬件抽象层模块后得到的句柄值。前面提到每一个硬件抽象层模块都对应一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用 dlopen 函数来加载与其对应的动态链接库文件的过程。在调用 dlclose 函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，我们在加载时，需要将它保存起来。

(5)　结构体 hw_module_t 的成员变量 methods 定义了一个硬件抽象层模块的操作方法列表，它的类型为 hw_module_methods_t，接下来，我们就介绍它的定义。

hw_module_methods_t 的定义代码如下所示：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t *module, const char *id,
            struct hw_device_t **device);
} hw_module_methods_t;
```

2.2.2 hw_module_methods_t

结构体 hw_module_methods_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t *module, const char *id,
            struct hw_device_t **device);

} hw_module_methods_t;
```

在结构体 hw_module_methods_t 中只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数 module 表示要打开的硬件设备所在的模块；参数 id 表示要打开的硬件设备的 ID；参数 device 是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此在调用结构体 hw_module_methods_t 的成员 open 打开一个硬件设备时，需要指定它的 ID。

2.2.3 hw_device_t

结构体 hw_device_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示：

```
typedef struct hw_device_t {
    /** tag must be initialized to HARDWARE_DEVICE_TAG */
    uint32 t tag;
    uint32_t version;

    /** reference to the module this device belongs to */
    struct hw_module_t *module;

    /** padding reserved for future use */
    uint32_t reserved[12];

    /** Close this device */
    int (*close)(struct hw_device_t *device);

} hw_device_t;
```

在 Android 系统中，硬件抽象层中的硬件设备使用结构体 hw_device_t 来描述。

在结构体 hw_device_t 中，需要注意如下所示的 3 点。

(1) 硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为 hw_device_t。

(2) 结构体 hw_device_t 的成员变量 tag 的值必须设置为 HARDWARE_DEVICE_TAG，即设置为一个常量值('H' <<24 | 'W'<<16 | 'D'<<8 | 'T')，用来标识这是一个硬件抽象层中的硬件设备结构体。

(3) 结构体 hw_device_t 的成员变量 close 是一个函数指针，它用来关闭一个硬件设备。

2.3　分析文件hardware.c

文件 hardware.c 是文件 hardware.h 的具体实现。本节内容中，将详细分析文件 hardware.c 的基本源码。

2.3.1　函数hw_get_module

函数 hw_get_module()能够根据模块 ID 寻找硬件模块动态链接库的地址，然后调用函数 load 打开动态链接库，并从中获取硬件模块结构体的地址。执行后，首先根据固定的符号 HAL_MODULE_INFO_SYM 寻找到结构体 hw_module_t，然后在 hw_module_t 中用 hw_module_methods_t 结构体成员的 open 函数打开相应的模块，并同时进行初始化操作。因为用户在调用 open()时通常都会传入一个指向 hw_device_t 指针的指针。这样函数 open()把对模块的操作结果保存到结构体 hw_device_t 中，用户通过它可以与模块进行交互。

函数 hw_get_module()的部分实现代码如下所示：

```
int hw_get_module(const char *id, const struct hw_module_t **module)
120 {
121     int status;
122     int i;
123     const struct hw_module_t *hmi = NULL;
124     char prop[PATH_MAX];
125     char path[PATH_MAX];
/* Loop through the configuration variants looking for a module */
135     for (i=0; i<HAL_VARIANT_KEYS_COUNT+1; i++)  {
```

2.3.2　数组variant_keys

在函数 hw_get_module()中需要用到数组 variant_keys，因为 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。定义此数组的代码如下所示：

```
* 44 static const char *variant_keys[] = {
* 45    "ro.hardware",  /* This goes first so that it can pick up a different
* 46                       file on the emulator. */
* 47    "ro.product.board",
* 48    "ro.board.platform",
* 49    "ro.arch"
* 50 };
```

然后通过此数组，使用如下代码得到操作权限：

```
136        if (i < HAL_VARIANT_KEYS_COUNT) {
137           if (property_get(variant_keys[i], prop, NULL) == 0) {
138              continue;
139           }
```

此处的 variant_keys[i]对应有三个值，分别是 trout、msm7k 和 ARMV6。

接下来，通过如下代码将路径和文件名保存到 path：

```
140  snprintf(path, sizeof(path), "%s/%s.%s.so",
141     HAL_LIBRARY_PATH, id, prop);
```

通过上述代码，把“HAL_LIBRARY_PATH/id.***.so”保存到 path 中，其中“***”就是上面 variant_keys 中各个元素所对应的值。

2.3.3　载入相应的库

载入相应的库，并把它们的 HMI 保存到 module 中。

具体代码如下所示：

```
142          } else {
143              snprintf(path, sizeof(path), "%s/%s.default.so",
144                      HAL_LIBRARY_PATH, id);
145          }
146          if (access(path, R_OK)) {
147              continue;
148          }
149          /* we found a library matching this id/variant */
150          break;
151      }
152
153      status = -ENOENT;
154      if (i < HAL VARIANT KEYS COUNT+1) {
155          /* load the module, if this fails, we're doomed, and we should not try
156           * to load a different variant. */
157          status = load(id, path, module); //load相应库。把它们的HMI保存到module中
158      }
159      return status;
161  }
```

2.3.4　打开相应库并获得hw_module_t结构体

打开相应的库，并获得 hw_module_t 结构体，具体代码如下所示：

```
60 static int load(const char *id,
61         const char *path,
62         const struct hw_module_t **pHmi)
63 {
64     int status;
65     void *handle;
```

```
66    struct hw_module_t *hmi;
      handle = dlopen(path, RTLD_NOW); //打开相应的库
74    if (handle == NULL) {
75        char const *err_str = dlerror();
76        LOGE("load: module=%s\n%s", path, err str?err str:"unknown");
77        status = -EINVAL;
78        goto done;
79    }

82    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
83    hmi = (struct hw_module_t *)dlsym(handle, sym); //获得hw_module_t结构体
84    if (hmi == NULL) {
85        LOGE("load: couldn't find symbol %s", sym);
86        status = -EINVAL;
87        goto done;
88    }
89
90    /* Check that the id matches */
91    if (strcmp(id, hmi->id) != 0) { //只是一个check
92        LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
93        status = -EINVAL;
94        goto done;
95    }
96
97    hmi->dso = handle;
98
99    /* success */
100   status = 0;
  done:
103   if (status != 0) {
104       hmi = NULL;
105       if (handle != NULL) {
106           dlclose(handle);
107           handle = NULL;
108       }
109   } else {
110       LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
111               id, path, *pHmi, handle);
112   }
113
114   *pHmi = hmi; //得到hw_module_t
115
116   return status;
117 }
```

2.4 分析硬件抽象层的加载过程

每一个硬件抽象层模块在内核中都对应有一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备的，它们是通过读写设备文件来进行通信的。硬件抽象层中的模块接口源文件一般保存在 hardware/libhardware 目录中，其目录结构如图 2-5 所示。

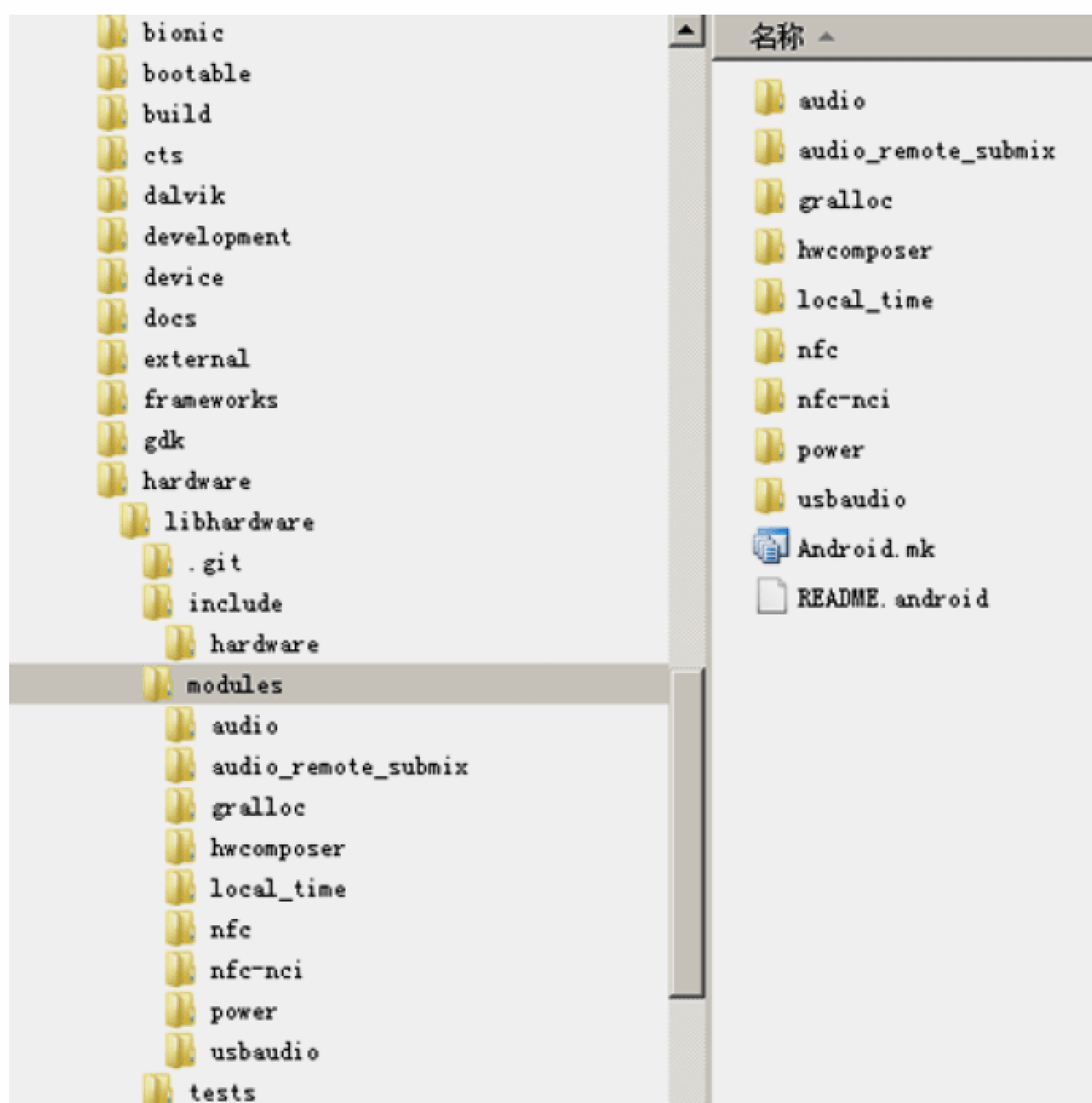


图 2-5 libhardware目录

Android 系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的 ID 值就可以了。在 Android 硬件抽象层中，负责加载硬件抽象层模块的函数是 hw_get_module，此函数在如下文件中定义：

```
hardware/libhardware/include/hardware/hardware.h
```

函数 hw_get_module 的实现原型如下：

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);
}
```

此函数有 id 和 module 两个参数。其中，id 是输入参数，表示要加载的硬件抽象层模块 ID；module 是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于 0，则表示加载成功；如果小于 0，则表示加载失败。

函数 hw_get_module 在文件 hardware/libhardware/hardware.c 中实现，具体的实现代码如下所示：

```
16 int hw_get_module(const char *id, const struct hw_module_t **module)
17 {
18    int status;
19    int i;
20    const struct hw_module_t *hmi = NULL;
21    char prop[PATH_MAX];
22    char path[PATH_MAX];
23
24 /*
25 * Here we rely on the fact that calling dlopen multiple times on
26 * the same .so will simply increment a refcount (and not load
27 * a new copy of the library).
28 * We also assume that dlopen() is thread-safe.
29 */
30
31 /* Loop through the configuration variants looking for a module */
32    for (i=0; i<HAL_VARIANT_KEYS_COUNT+1; i++) {
33        if (i < HAL_VARIANT_KEYS_COUNT) {
34            if (property_get(variant_keys[i], prop, NULL) == 0) {
35                continue;
36            }
37
38            snprintf(path, sizeof(path), "%s/%s.%s.so",
39                HAL_LIBRARY_PATH1, id, prop);
40            if (access(path, R_OK) == 0) break;
41
42            snprintf(path, sizeof(path), "%s/%s.%s.so",
43                HAL_LIBRARY_PATH2, id, prop);
44            if (access(path, R_OK) == 0) break;
45            } else {
46                snprintf(path, sizeof(path), "%s/%s.default.so",
47                         HAL_LIBRARY_PATH1, id);
48                if (access(path, R_OK) == 0) break;
49            }
50        }
51
52        status = -ENOENT;
53        if (i < HAL_VARIANT_KEYS_COUNT+1) {
54 /* load the module, if this fails, we're doomed, and we should not try
55 * to load a different variant. */
56            status = load(id, path, module);
57       }
58
59       return status;
60 }
```

在上述代码中，数组 variant_keys 用来组装要加载的硬件抽象层模块的文件名称。常量 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。宏 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 用来定义要加载的硬件抽象层模块文件所在的目录。第 32 行到第 50 行的 for 循环根据数组 variant_keys 在 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 目录中检查对应的硬件抽象层模块文件是否存在，如果存在，则结束 for 循环；第 56 行调用 load 函数来执行加载硬件抽象层模块的操作。

编译好的模块文件位于 out/target/product/generic/system/lib/hw 目录中，而这个目录经过打包后，就对应于设备上的/system/lib/hw 目录。

宏 HAL_LIBRARY_PATH2 所定义的目录为/vendor/lib/hw，用来保存设备厂商所提供的硬件抽象层模块接口文件。

在上述第 56 行代码中，调用函数 load 执行硬件抽象层模块的加载操作，此函数的具体实现代码如下所示：

```
01 static int load(const char *id,
02   const char *path,
03   const struct hw_module_t **pHmi)
04 {
05     int status;
06     void *handle;
07     struct hw_module_t *hmi;
08
09 /*
10 * load the symbols resolving undefined symbols before
11 * dlopen returns. Since RTLD_GLOBAL is not or'd in with
12 * RTLD_NOW the external symbols will not be global
13 */
14     handle = dlopen(path, RTLD_NOW);
15     if (handle == NULL) {
16         char const *err_str = dlerror();
17         LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
18         status = -EINVAL;
19         goto done;
20     }
21
22 /* Get the address of the struct hal_module_info. */
23     const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
24     hmi = (struct hw_module_t *)dlsym(handle, sym);
25     if (hmi == NULL) {
26         LOGE("load: couldn't find symbol %s", sym);
27         status = -EINVAL;
28         goto done;
29     }
30
```

```
31 /* Check that the id matches */
32     if (strcmp(id, hmi->id) != 0) {
33         LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
34         status = -EINVAL;
35         goto done;
36     }
37 
38     hmi->dso = handle;
39 
40     /* success */
41     status = 0;
42 
43 done:
44     if (status != 0) {
45         hmi = NULL;
46         if (handle != NULL) {
47             dlclose(handle);
48             handle = NULL;
49         }
50     } else {
51         LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
52                 id, path, *pHmi, handle);
53     }
54 
55     *pHmi = hmi;
56 
57     return status;
58 }
```

在上述代码中，第 14 行调用函数 dlopen 将模块加载到内存中。加载完成这个动态链接库文件后，第 24 行就调用函数 dlsym 来获得里面名称为 HAL_MODULE_INFO_SYM_AS_STR 的符号。这个 HAL_MODULE_INFO_SYM_AS_STR 符号指向的是一个自定义的硬件抽象层模块结构体，它包含了对应的硬件抽象层模块的所有信息。

HAL_MODULE_INFO_SYM_AS_STR 是一个宏，它的值定义为“HMI”。

根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为 HMI 的符号，而且这个符号的第一个成员变量的类型必须定义为 hw_module_t，因此，第 24 行可以安全地将模块中的 HMI 符号转换为一个 hw_module_t 结构体指针。

获得了这个 hw_module_t 结构体指针之后，第 32 行调用 strcmp 函数来验证加载得到的硬件抽象层模块 ID 是否与所要求加载的硬件抽象层模块 ID 一致。如果不一致，就说明出错了，函数返回一个错误值-EINVAL。

最后，第 38 行将成功加载后，得到的模块句柄值 handle 保存在 hw_module_t 结构体指针 hmi 的成员变量 dso 中，然后将它返回给调用者。

2.5 分析硬件访问服务

开发好硬件抽象层模块之后，我们通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用C++语言开发的，而应用程序框架层中的硬件访问服务是使用Java语言开发的，因此，硬件访问服务必须通过Java本地接口(Java Native Interface，JNI)来调用硬件抽象层模块的接口。

Android系统的硬件访问服务通常运行在系统进程System5中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间的通信机制来访问这些硬件访问服务。Android系统提供了一种高效的进程间通信机制——Binder进程间通信机制6，应用程序就是通过它来访问运行在系统进程System中的硬件访问服务的。Binder进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，我们首先要定义它的服务接口。

2.5.1 定义硬件访问服务接口

Android系统提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为Android接口描述语言(Android Interface Definition Language，AIDL)。以AIDL定义的服务接口文件是以aidl为后缀名的，在编译时，编译系统会将它们转换成一个Java文件，然后再对它们进行编译。在本节中，我们将使用AIDL来定义硬件访问服务接口IFregService。

在Android系统中，通常把硬件访问服务接口定义在frameworks/base/core/java/android/os目录中，所以把定义了硬件访问服务接口IFregService的文件IFregService.aidl也保存在这个目录中，其具体内容如下所示：

```
package android.os;
interface IFregService {
    void setVal(int val);
    int getVal();
}
```

服务接口IFregService只定义了两个成员函数，它们分别是setVal和getVal。其中，成员函数setVal用来往虚拟硬件设备freg的寄存器val中写入一个整数，而成员函数getVal用来从虚拟硬件设备freg的寄存器val中读出一个整数。

由于服务接口IFregService是使用AIDL语言描述的，因此需要将其添加到编译脚本文件中，这样编译系统才能将其转换为Java文件，然后再对它进行编译。进入到frameworks/base目录中，打开里面的Android.mk文件，修改LOCAL_SRC_FILES变量的值：

```
LOCAL SRC FILES += \
...
voip/java/android/net/sip/ISipService.aidl \
core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，我们就可以使用mmm命令对硬件访问服务接口IFregService

进行编译了：

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的 framework.jar 文件就包含有 IFregService 接口，它继承了 android.os.IInterface 接口。在 IFregService 接口内部，定义了一个 Binder 本地对象类 Stub，它实现了 IFregService 接口，并且继承了 android.os.Binder 类。此外，在 IFregService.Stub 类内部，还定义了一个 Binder 代理对象类 Proxy，它同样也实现了 IFregService 接口。

用 AIDL 定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为 Server 进程，而使用服务的进程称为 Client 进程。在 Server 进程中，每一个服务都对应有一个 Binder 本地对象，它通过一个桩(Stub)来等待 Client 进程发送进程间通信请求。Client 进程在访问运行 Server 进程中的服务之前，首先要获得它的一个 Binder 代理对象接口(Proxy)，然后通过这个 Binder 代理对象接口向它发送进程间通信请求。

2.5.2　实现硬件访问服务

在 Android 系统中，因为通常通过 frameworks/base/services/java/com/android/server 目录中的文件实现硬件访问服务，所以把实现了硬件访问服务 FregService 的文件 FregService.java 也保存在这个目录中，其具体代码如下所示：

```
01 package com.android.server;
02
03 import android.content.Context;
04 import android.os.IFregService;
05 import android.util.Slog;
06
07 public class FregService extends IFregService.Stub {
08    private static final String TAG = "FregService";
09
10    private int mPtr = 0;
11
12    FregService() {
13       mPtr = init_native();
14
15       if(mPtr == 0) {
16          Slog.e(TAG, "Failed to initialize freg service.");
17       }
18    }
19
20    public void setVal(int val) {
21       if(mPtr == 0) {
22          Slog.e(TAG, "Freg service is not initialized.");
23          return;
24       }
25
26       setVal_native(mPtr, val);
27    }
```

```
28
29    public int getVal() {
30        if(mPtr == 0) {
31            Slog.e(TAG, "Freg service is not initialized.");
32            return 0;
33        }
34
35        return getVal_native(mPtr);
36    }
37
38    private static native int init_native();
39    private static native void setVal_native(int ptr, int val);
40    private static native int getVal_native(int ptr);
41 };
```

在上述代码中，硬件访问服务 FregService 继承了类 IFregService.Stub，并且实现了 IFregService 接口的成员函数 setVal 和 getVal。其中，成员函数 setVal 通过调用 JNI 方法 setVal_native 来写虚拟硬件设备 freg 的寄存器 val，而成员函数 getVal 调用 JNI 方法 getVal_native 来读虚拟硬件设备 freg 的寄存器 val。在启动硬件访问服务 FregService 时，会通过调用 JNI 函数 init_native 来打开虚拟硬件设备 freg，并且获得它的一个句柄值，保存在成员变量 mPtr 中。如果硬件访问服务 FregService 打开虚拟硬件设备 freg 失败，那么它的成员变量 mPtr 的值就等于 0；否则，就得到一个大于 0 的句柄值。这个句柄值实际上是指向虚拟硬件设备 freg 在硬件抽象层中的一个设备对象，硬件访问服务 FregService 的成员函数 setVal 和 getVal 在访问虚拟硬件设备 freg 的寄存器 val 时，必须指定这个句柄值，以便硬件访问服务 FregService 的 JNI 实现可以知道它所要访问的是哪一个硬件设备。

2.6　分析mokoid工程

在 Android 4.3 源码的配套资料中，在 mokoid 工程中提供了一个 LedTest 示例来演示 HAL 层的执行过程，此工程演示了 Android 层次结构和 HAL 的编程方法。LedTest 示例程序源码可以从网络中获取，下面是在 Linux 系统中的下载获取此工程的命令：

```
#svn checkout http://mokoid.googlecode.com/svn/trunk/mokoid-read-only
```

下载 mokoid 工程文件后，其目录结构如图 2-6 所示。

在 Android 中，需要通过 JNI(Java Native Interface)来实现 Android 的 HAL，JNI 就是 Java 程序可以调用 C/C++写的动态链接库，所以 HAL 可以使用 C/C++语言编写，这样做的好处是效率更高。在 Android 系统中有如下两种访问 HAL 的方式。

(1)　Android 应用程序直接通过 Service 调用“.so”格式的 JNI：此方法比较简单高效，但是不正规。

(2)　经过 Manager 调用 Service：此方法实现起来比较复杂，但更符合目前的 Android 框架。在此方法中，在进程 LedManager 和 LedService(Java)中需要通过进程通信的方式实现通信。

mokoid 工程中分别实现了上述两种方法，接下来将详细介绍这两种方法的具体实现原理。

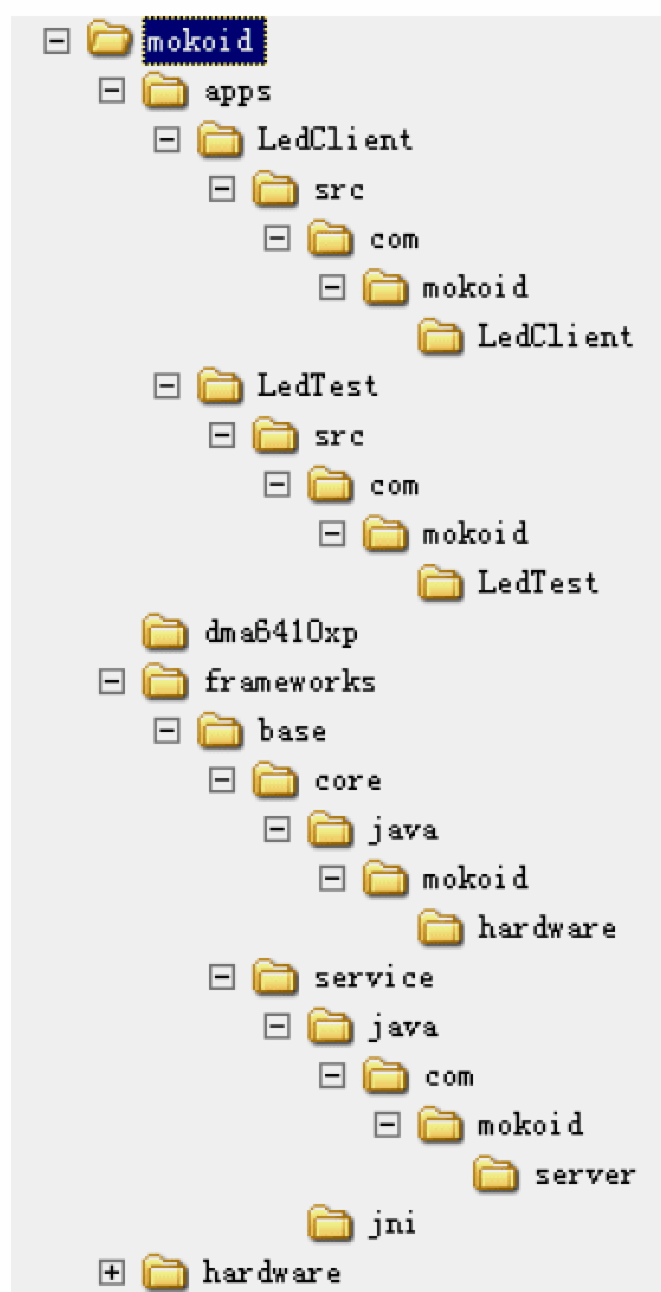


图 2-6　mokoid工程的目录结构

2.6.1　直接调用Service方法实现

(1)　HAL 层的实现代码

文件 hardware/modules/led/led.c 的实现代码如下所示：

```
#define LOG_TAG "MokoidLedStub"
#include <hardware/hardware.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#include <mokoid/led.h>
/****************************************************************************/
int led_device_close(struct hw_device_t *device)
{
    struct led_control_device_t *ctx = (struct led_control_device_t*)device;
    if (ctx) {
        free(ctx);
    }
    return 0;
}
int led_on(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d on.", led);
    return 0;
}
```

```
int led_off(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d off.", led);
    return 0;
}
static int led_device_open(const struct hw_module_t *module, const char *name,
      struct hw_device_t **device)
{
    struct led_control_device_t *dev;
    dev = (struct led_control_device_t*)malloc(sizeof(*dev));
    memset(dev, 0, sizeof(*dev));
    dev->common.tag =  HARDWARE_DEVICE_TAG;
    dev->common.version = 0;
    dev->common.module = module;
    dev->common.close = led_device_close;
    dev->set_on = led_on; //实例化支持的操作
    dev->set_off = led_off;
    *device = &dev->common; //将实例化的 led_control_device_t 地址返回给 JNI 层
success:
    return 0;
}
static struct hw_module_methods_t led_module_methods = {
   open: led_device_open
};
const struct led_module_t HAL_MODULE_INFO_SYM = {
   //定义此对象相当于向系统注册了一个 ID 为 LED_HARDWARE_MODULE_ID 的 Stub
   common: {
      tag: HARDWARE_MODULE_TAG,
      version_major: 1,
      version_minor: 0,
      id: LED_HARDWARE_MODULE_ID,
      name: "Sample LED Stub",
      author: "The Mokoid Open Source Project",
      methods: &led_module_methods, //实现了一个 open 的方法供 JNI 层调用
   }
   /* supporting APIs go here */
};
```

(2) JNI 层的实现代码

文件 frameworks/base/service/jni/com_mokoid_server_LedService.cpp 的实现代码如下所示：

```
struct led_control_device_t *sLedDevice = NULL;
static jboolean mokoid_setOn(JNIEnv *env, jobject thiz, jint led) {
   LOGI("LedService JNI: mokoid_setOn() is invoked.");
   if (sLedDevice == NULL) {
      LOGI("LedService JNI: sLedDevice was not fetched correctly.");
      return -1;
   } else {
      return sLedDevice->set_on(sLedDevice, led); //调用 HAL 层的注册方法
```

```
    }
}
static jboolean mokoid_setOff(JNIEnv *env, jobject thiz, jint led) {
    LOGI("LedService JNI: mokoid_setOff() is invoked.");
    if (sLedDevice == NULL) {
        LOGI("LedService JNI: sLedDevice was not fetched correctly.");
        return -1;
    } else {
        return sLedDevice->set_on(sLedDevice, led); //调用 HAL 层的注册方法
    }
}
/** helper APIs——JNI 通过 LED_HARDSOFTWARE_MODULE_ID 找到对应的 Stub */
static inline int led_control_open(const struct hw_module_t *module,
        struct led_control_device_t **device) {
    return module->methods->open(module,
            LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
}
static jboolean
mokoid_init(JNIEnv *env, jclass clazz) {
    led_module_t *module;
    if (hw_get_module(LED_HARDWARE_MODULE_ID,
      (const hw_module_t**)&module) == 0) {
        LOGI("LedService JNI: LED Stub found.");
        if (led_control_open(&module->common, &sLedDevice) == 0) {
            LOGI("LedService JNI: Got Stub operations.");
            return 0;
        }
    }
    LOGE("LedService JNI: Get Stub operations failed.");
    return -1;
}
/*
 * Array of methods.
 *
 * Each entry has three fields: the name of the method, the method
 * signature, and a pointer to the native implementation.
 */
// JNINativeMethod 是 JNI 层的注册方法
static const JNINativeMethod gMethods[] = {
    {"_init",        "()Z", //Framework 层调用_init 时触发
            (void*)mokoid_init},
    { "_set_on",     "(I)Z",    (void*)mokoid_setOn },
    { "_set_off",    "(I)Z",   (void*)mokoid_setOff },
};
static int registerMethods(JNIEnv *env) {
    static const char *const kClassName = "com/mokoid/server/LedService";
    jclass clazz;
    /* look up the class */
    clazz = env->FindClass(kClassName);
```

```
    if (clazz == NULL) {
        LOGE("Can't find class %s\n", kClassName);
        return -1;
    }
    /* register all the methods */
    if (env->RegisterNatives(clazz, gMethods,
            sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
    {
        LOGE("Failed registering methods for %s\n", kClassName);
        return -1;
    }
    /* fill out the rest of the ID cache */
    return 0;
}
// ------------------------------------------------------------------------

/*
 * This is called by the VM when the shared library is first loaded.
 */
////Framework 层加载 JNI 库时调用
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
    JNIEnv *env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**)&env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    if (registerMethods(env) != 0) {
        LOGE("ERROR: PlatformLibrary native registration failed\n");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

(3) Service 的实现代码

这里的 Service 属于 Framework 层，实现文件是 LedService.java，被保存在如下所示的目录中：

```
frameworks\base\service\java\com\mokoid\server
```

LedService.java 的具体实现代码如下所示：

```
package com.mokoid.server;

import android.util.Config;
import android.util.Log;
import android.content.Context;
```

```
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.os.IBinder;
import mokoid.hardware.ILedService;

public final class LedService extends ILedService.Stub {

    static {
        System.load("/system/lib/libmokoid_runtime.so"); //加载 JNI 动态库
    }

    public LedService() {
        Log.i("LedService", "Go to get LED Stub...");
        _init();
    }

    /*
     * Mokoid LED native methods.
     */
    public boolean setOn(int led) {
        Log.i("MokoidPlatform", "LED On");
        return _set_on(led);
    }
    public boolean setOff(int led) {
        Log.i("MokoidPlatform", "LED Off");
        return _set_off(led);
    }
    private static native boolean _init(); //声明 JNI 库可以提供的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
}
```

(4)　APP 测试程序的实现代码

此处的测试程序属于 APP 层，文件 apps/LedClient/src/com/mokoid/LedClient/LedClient.java 的实现代码如下所示：

```
package com.mokoid.LedClient;
import com.mokoid.server.LedService; //导入 Framework 层的 LedService

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class LedClient extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        // Call an API on the library.
        LedService ls = new LedService(); //实例化LedService
        ls.setOn(1); //通过LedService提供的方法控制底层硬件

        TextView tv = new TextView(this);
        tv.setText("LED 0 is on.");
        setContentView(tv);
    }
}
```

2.6.2 通过Manager调用Service实现

(1) Manager 的实现代码

应用程序通过此 Manager 和 Service 实现通信功能，对应文件 frameworks/base/core/java/mokoid/hardware/LedManager.java 的实现代码如下所示：

```
package mokoid.hardware;

import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.ParcelFileDescriptor;
import android.os.Process;
import android.os.RemoteException;
import android.os.Handler;
import android.os.Message;
import android.os.ServiceManager;
import android.util.Log;
import mokoid.hardware.ILedService;

/**
 * Class that lets you access the Mokoid LedService.
 */
public class LedManager {
    private static final String TAG = "LedManager";
    private ILedService mLedService;

    public LedManager() {

        mLedService = ILedService.Stub.asInterface(
                         ServiceManager.getService("led"));

        if (mLedService != null) {
            Log.i(TAG, "The LedManager object is ready.");
        }
    }
```

```
    public boolean LedOn(int n) {
        boolean result = false;

        try {
            result = mLedService.setOn(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOn:", e);
        }
        return result;
    }

    public boolean LedOff(int n) {
        boolean result = false;

        try {
            result = mLedService.setOff(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOff:", e);
        }
        return result;
    }
}
```

因为 LedService 和 LedManager 分别属于不同的进程，所以在此需要考虑不同进程之间的通信问题。此时在 Manager 中可以增加一个 aidl 文件来描述通信接口，文件 frameworks/base/core/java/mokoid/hardware/ILedService.aidl 的实现代码如下所示：

```
package mokoid.hardware;

interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
```

(2)　SystemServer 的实现代码

SystemServer 属于 APP 层，其 apps/LedTest/src/com/mokoid/LedTest/LedSystemServer.java 实现文件的主要代码如下所示：

```
package com.mokoid.LedTest;

import com.mokoid.server.LedService;

import android.os.IBinder;
import android.os.ServiceManager;
import android.util.Log;
import android.app.Service;
import android.content.Context;
import android.content.Intent;
```

```
public class LedSystemServer extends Service { //代表一个后台进程
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    public void onStart(Intent intent, int startId) {
        Log.i("LedSystemServer", "Start LedService...");
        /* Please also see SystemServer.java for your interests. */
        LedService ls = new LedService();
        try {
            ServiceManager.addService("led", ls);
        } catch (RuntimeException e) {
            Log.e("LedSystemServer", "Start LedService failed.");
        }
    }
}
```

(3) APP 测试程序

此处的测试程序属于 APP 层，文件 mokoid-read-only/apps/LedTest/src/com/mokoid/LedTest/LedTest.java 的实现代码如下所示：

```
package com.mokoid.LedTest;
import mokoid.hardware.LedManager;
import com.mokoid.server.LedService;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Button;
import android.content.Intent;
import android.view.View;

public class LedTest extends Activity implements View.OnClickListener {
    private LedManager mLedManager = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Start LedService in a seperated process.
        startService(new Intent("com.mokoid.systemserver"));
        // Just for testing. !! PLEASE DON't DO THIS !!
        //LedService ls = new LedService();
        Button btn = new Button(this);
        btn.setText("Click to turn LED 1 On");
        btn.setOnClickListener(this);
        setContentView(btn);
    }
    public void onClick(View v) {
```

```
        // Get LedManager.
        if (mLedManager == null) {
            Log.i("LedTest", "Creat a new LedManager object.");
            mLedManager = new LedManager();
        }
        if (mLedManager != null) {
            Log.i("LedTest", "Got LedManager object.");
        }
       /** Call methods in LedService via proxy object
        * which is provided by LedManager.
        */
        mLedManager.LedOn(1);
        TextView tv = new TextView(this);
        tv.setText("LED 1 is On.");
        setContentView(tv);
    }
}
```

2.7　分析HAL层的具体实现(以Sensor系统为例)

为了了解 Android 4.3 系统中 HAL 层的具体实现，本节将以 Sensor 传感器系统为例，介绍 HAL 的具体实现过程，为读者步入本书后面知识的学习打下基础。

2.7.1　传感器系统的基础知识

人们在日常生活中，会经常用到传感器，例如楼宇的楼梯灯、马路上的路灯等。在 Android 手机中提供了加速度传感器，以及磁场、方向、陀螺仪、光线、压力、温度等传感器。

在 Android 系统中，传感器的代码分布信息如下所示。

(1)　传感器系统的 Java 部分，实现文件为 Sensor*.java，代码路径为：

```
frameworks/base/include/core/java/android/hardware
```

(2)　传感器系统的 JNI 部分，此部分演示了 android.hardware.Sensor.Manager 类的本质支持。代码路径为：

```
frameworks/base/core/jni/android_hardware_SensorManager.cpp
```

(3)　传感器系统的 HAL 层，演示了传感器系统的硬件抽象层所需要的具体实现。头文件路径为：

```
hardware/libhardware/include/hardware/sensors.h
```

(4)　驱动层，其实现代码路径为：

```
kernel/driver/hwmon/$(PROJECT)/sensor
```

在本节后面的内容中，将详细分析上述源码文件的具体实现过程。

2.7.2 HAL层的Sensor代码

(1) 文件 Android.mk

HAL 层的代码都是“c/cpp”格式，一般保存在目录 hardware/$(PROJECT)/sensor/中。文件 Android.mk 的主要代码如下所示：

```
LOCAL_PATH:= $(call my-dir)
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL SHARED LIBRARIES := libcutils libc liblog
LOCAL_SRC_FILES := 适配文件
LOCAL_MODULE := sensors.$(PROJECT)
include $(BUILD_SHARED_LIBRARY)
```

在这里要注意 LOCAL_MODULE 的赋值，这里的模块名字都是预先定义好的，具体可以参考 hardware/libhardware/hardware.c。

这里也可以看到加载的顺序，在加载 Sensor 的时候，依次去查找这些“so”是不是存在，然后就可以把加载对应到适配。

(2) 填充的结构体

在 HAL 层中需要注意下面的填充结构体。

① 定义 sensor 模块的代码如下所示：

```
struct sensor_module_t {
    struct hw_module_t common;
    int (*get_sensors_list)(struct sensors_module_t *module,
      struct sensor_t const **list);
};
```

其中的 get_sensors_list()表示用来获得传感器列表。

② 用 sensor_t 表示一个传感器的描述，具体代码如下所示：

```
struct sensor t {
    const char *name; //传感器名称
    const char *vendor; //传感器的 Vendor
    int version; //传感器的版本
    int handle; //传感器的句柄
    int type; //传感器类型
    float maxRange; //传感器的最大范围
    float resolution; //传感器的解析度
    float power; //传感器的功耗
    void *reserved[9];
}
```

③ 定义结构体和共用体的代码如下所示：

```
typedef struct {
    int sensor; //sensor 标识符
    union {
```

```
        sensors_vec_t vector; //x,y,z 矢量
        sensors_vec_t orientation; //方向值，单位为角度
        sensors_vec_t acceleration; //加速度值，单位为 m/s2
        sensors_vec_t magnetic; //磁矢量，单位为 uT
        float temperature; //温度，单位为℃
        float distance; //距离，单位为 cm
        float light; //光线亮度，单位为 lux
    }
    int64_t time; //ns
    uint32_t reserved;
} sensors_data_t;
```

(3)　适配层函数接口

在 HAL 层中需要注意下面这个函数：

```
static int s_device_open(const struct hw_module_t *module,
            const char *name,
            struct hw_device_t **device)
```

其中要特别注意下面的赋值：

```
if (!strcmp(name, SENSORS_HARDWARE_CONTROL)) { //命令通路
    ....
    dev->device.common.close = dev_control_close;
    dev->device.open_data_source = open_data_source;
    dev->device.activate = activate;
    dev->device.set delay = set delay;
    ...
} else if (!strcmp(name, SENSORS_HARDWARE_DATA)) { //数据通路
    ...
    dev->device.common.close = dev_data_close;
    dev->device.data_open = data_open;
    dev->device.data_close = data_close;
    dev->device.poll = poll;
    ...
}
```

在驱动中提供的代码要实现 file_operation。在驱动代码中获得的 Sensor 寄存器中的值并不一定是我们实际要上报给应用的值，比如 g-sensor，则各个方向不应该大于 10。

2.7.3　Sensor编程的流程

Sensor 编程的基本流程如下所示。

(1)　获取系统服务(SENSOR_SERVICE)，返回一个 SensorManager 对象：

```
sensormanager = (SensorManager)getSystemSeriver(SENSOR_SERVICE);
```

(2)　通过 SensorManager 对象获取相应的 Sensor 类型的对象：

```
sensorObject = sensormanager.getDefaultSensor(sensor Type);
```

(3) 声明一个 SensorEventListener 对象，用于侦听 Sensor 事件，并重载 onSensorChanged 方法：

```
SensorEventListener sensorListener = new SensorEventListener() {};
```

(4) 注册相应的 SensorService：

```
sensormanager.registerListener(sensorListener, sensorObject, Sensor TYPE);
```

(5) 销毁相应的 SensorService：

```
sensormanager.unregisterListener(sensorListener, sensorObject);
```

此处的 SensorListener 接口是整个传感器应用程序的核心，它包括如下两个必需的方法。

onSensorChanged(int sensor,float values[])：此方法在传感器值更改时调用，该方法只对受此应用程序监视的传感器调用。该方法包括如下两个参数：

- 一个整数：指示更改的传感器。
- 一个浮点值数组：表示传感器数据本身。

注意： 有些传感器只提供一个数据值，另一些则提供三个浮点值。方向和加速度传感器都提供三个数据值。

onAccuracyChanged(int sensor, int accuracy)：当传感器的准确性更改时，将调用此方法，此方法的参数包括两个整数，一个表示传感器，另一个表示该传感器新的准确值。

第 3 章
分析JNI(Java本地接口)层

JNI 是 Java Native Interface 的缩写，译为 Java 本地接口。JNI 标准是 Java 平台的一部分，允许 Java 代码与其他语言编写的代码进行交互。

因为 JNI 是本地编程接口，所以只要是能够在 Java 虚拟机(VM)内部运行的 Java 代码，就都能够与用其他编程语言(如 C、C++和汇编语言)编写的应用程序和库进行交互操作。另外，为了讲解 JNI 在 Java 和 C/C++之间的作用，还将特意讲解 JNI 在 media 框架中的应用。

在本章的内容中，将详细分析 Android 4.3 系统的 JNI 层的基本知识，为读者步入本书后面知识的学习打下基础。

3.1 JNI基础

在 Android 系统中，JNI 是连接 Java 部分和 C/C++部分的纽带。要想完整地使用 JNI，需要仔细分析 Java 代码和 C/C++代码。在 Android 中通过提供 JNI 的方式，让 Java 程序可以调用 C 语言程序。Android 中的很多 Java 类都具有 Native(本地)接口，这些接口由本地代码实现，然后注册到系统中。

3.1.1 JNI的层次结构

JNI 调用的层次非常清晰，主要分为三层，在 Android 系统中，这三层从上到下依次为：Java → JNI → C/C++(SO 库)，Java 可以访问 C/C++中的方法，同样 C/C++可以修改 Java 对象，图 3-1 清晰地描述了这三者之间的调用关系。

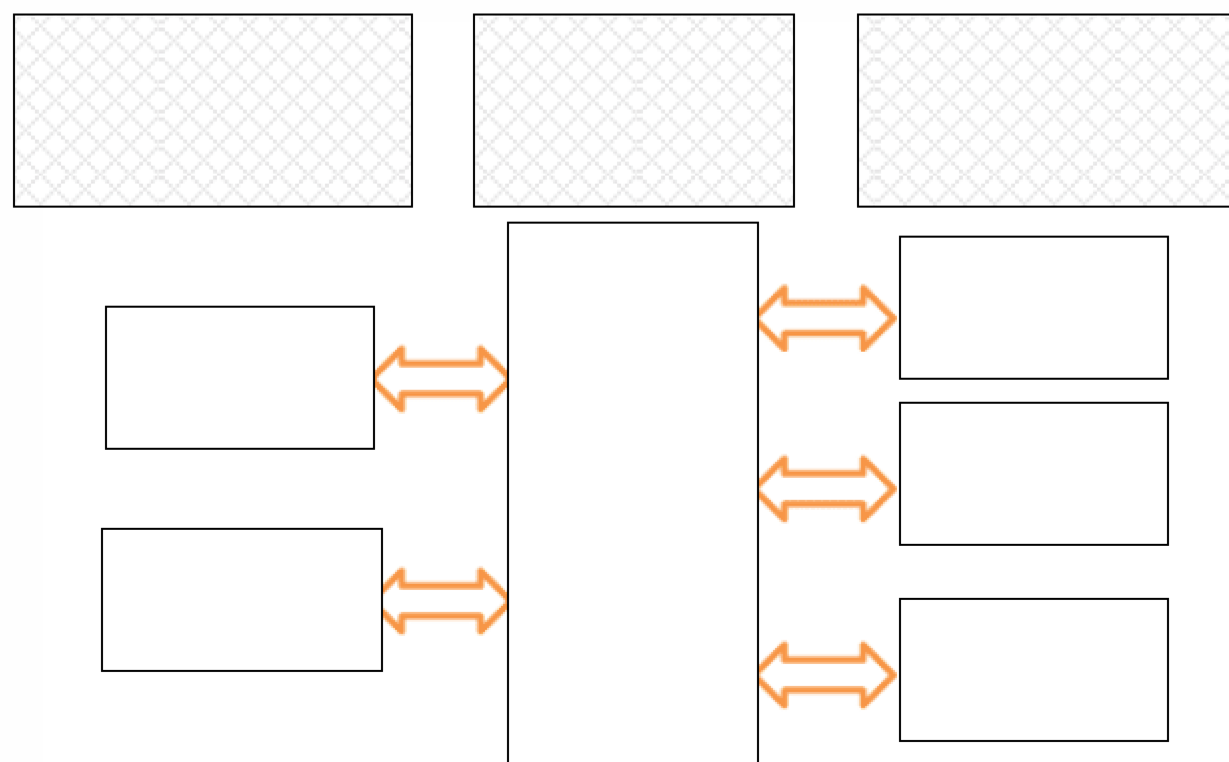


图 3-1 JNI调用的层次关系

由图 3-1 可知，JNI 的调用关系为：

```
Java-----------------------JNI-----------------------Native
```

在 Android 4.3 的源码中，主要的 JNI 代码放在以下的路径中：

```
frameworks/base/core/jni/
```

上述路径中的内容被编译成 libandroid_runtime.so 库，这是一个普通的动态库，被放置在目标系统的/system/lib 目录下。另外，Android 中还存在其他的 JNI 库，其实 JNI 中的各个文件就是普通的 C++源文件；在 Android 中实现的 JNI 库，需要连接动态库 libnativehelper.so。

3.1.2 JNI的本质

C和C++实现
（属于系统底层）

属于中

从本质上来说，Java 语言的运行完全依赖于脚本引擎对 Java 的代码进行解释和执行。因为现代的 Java 可以从源代码编译成.class 之类的中间格式的二进制文件，所以这种处理会加快 Java 脚本的运行速度。尽管如此，基本的执行方式仍然不变，由脚本引擎(被称为 JVM)来执行。

Functions
各种定义

与 Python、Perl 之类的纯脚本相比，只是把脚本变成了二进制格式而已。另外，Java 本身就是一门对面向对象语言，可以调用完善的功能库。当把这个脚本引擎移植到所有平台上之后，那么这个脚本就很自然地实现“跨平台”了。绝大多数的脚本引擎都支持一个很显著的特性，就是可以通过 C/C++编写模块，并在脚本中调用这些模块。同样 Java 也是如此，Java 一定要提供一种在脚本中调用 C/C++编写的模块的机制，才能称得上是一个相对完善的脚本引擎。

从本质上来看，Android 平台是由 arm-linux 操作系统和一个叫作 Dalvik 的 Java 虚拟机组成的。所有在 Android 模拟器上面看到的界面效果都是用 Java 语言编写的，具体请看源代码中的 frameworks/base 目录。由此可见，Dalvik 只是提供了一个标准的支持 JNI 调用的 Java 虚拟机环境。在 Android 平台中，使用 JNI 技术封装了所有的与硬件相关的操作，通过 Java 去调用 JNI 模块，而 JNI 模块使用 C/C++调用 Android 本身的 arm-linux 底层驱动，这样便实现了对硬件的调用。

3.1.3　与JNI相关的文件

在 Android 4.3 的源码中，与 JNI 相关的文件如下所示：

```
./frameworks/base/media/java/android/media/MediaScanner.java
./frameworks/base/media/jni/android_media_MediaScanner.cpp
./frameworks/base/media/jni/android_media_MediaPlayer.cpp
./frameworks/base/media/jni/AndroidRuntime.cpp
./libnativehelper/JNIHelp.cpp
```

由此可见，与 JNI 密切相关的是 Media 系统，而 Media 系统的架构基础是 MediaScanner。在启动 Android 系统之初，就会扫描出系统中的 Media 文件供后续应用使用，既有新加入的媒体，也有几微秒种前删除的媒体文件，并且还需要自动更新相应的媒体库。在 Android 系统中，与用户体验密切相关的 Music、Gallery 播放等应用，也是基于 MediaScanner 的扫描媒体文件功能的。MediaScanner 位于 Android 4.3 源码的如下路径中：

```
packages/providers/MediaProvider
```

上述路径包含了三个主要部分，分别是 MediaScannerReceiver、MediaScannerService 和 MediaProvider。在 MediaProvider 目录下的 AndroidManifest 中可以查看 MediaProvider 的基本架构，如图 3-2 所示。

- MediaScannerReceiver：是一个 BroadcastReceiver(广播接收)，功能是进行媒体扫描，这也是 MediaScanner 提供给外界的接口之一。收到广播之后启动 MediaScannerService 具体执行扫描工作。
- MediaScannerService：是一个 Service，负责媒体扫描，它还要用到 Framework 中的 MediaScanner 来共同完成具体的扫描工作，扫描的结果在 MediaProvider 提供的数据库中。
- MediaProvider：是一个 ContentProvider，媒体库(Images/Audio/Video/Playlist 等)的数据提供者。负责操作数据库，并提供给别的程序 insert、query、delete、update 等操作。

在本章接下来的内容中，以 MediaScanner 源码分析作为基础，将详细分析 JNI 在 Android 系统中的作用。

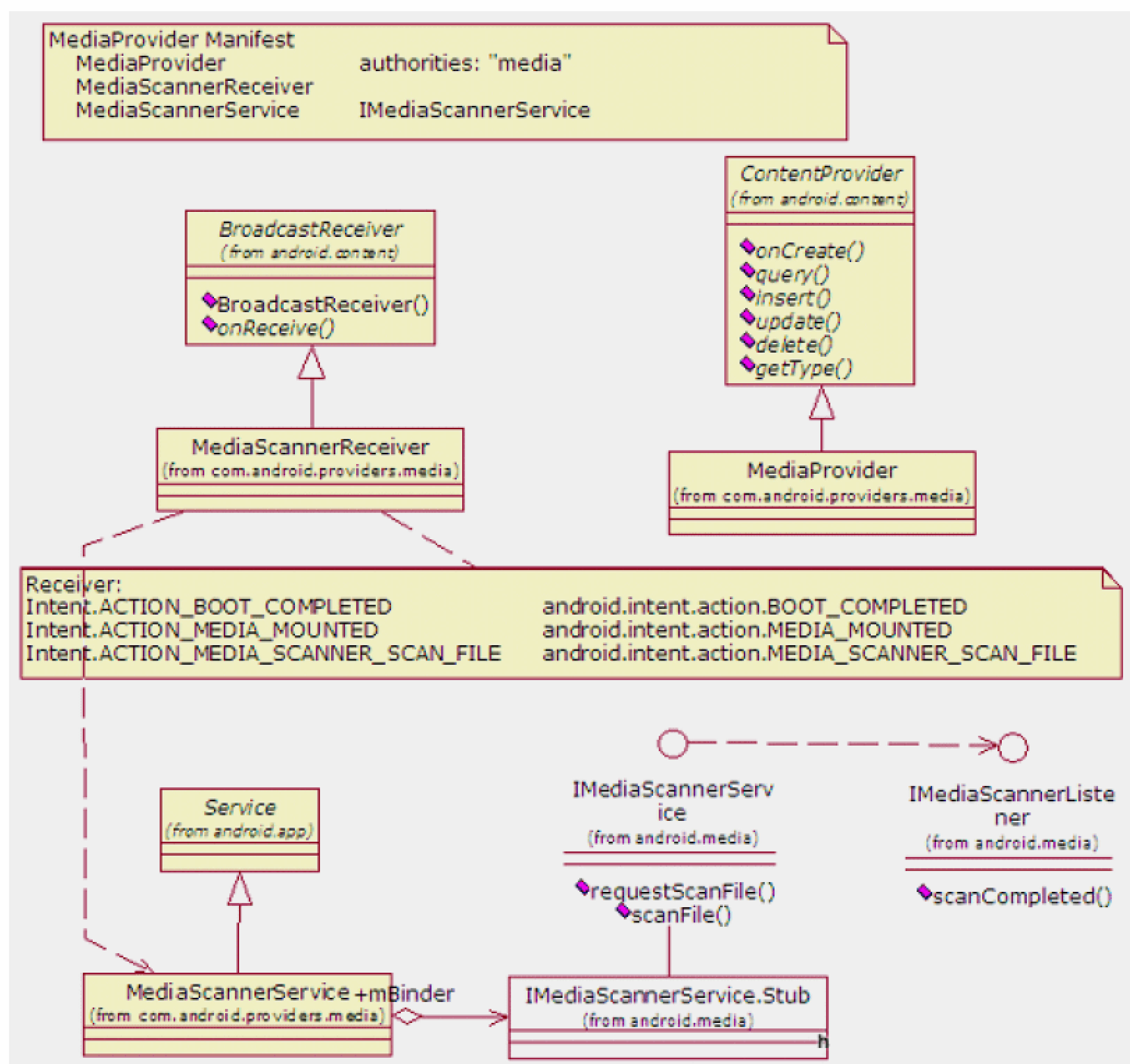


图 3-2　MediaProvider的基本架构

3.2　分析Java层

在 MediaScanner 系统中，JNI 的调用关系为：

```
MediaScanner-------------libmedia_jni.so-------------libmedia.so
```

在 Android 系统中，MediaScanner 的功能是扫描媒体文件，得到诸如歌曲时长、歌曲作者等信息，并将这些信息存放到媒体数据库中，以供其他应用程序使用。

在 JNI 应用中，Java 层 MediaScanner 的实现文件为：

```
./frameworks/base/media/java/android/media/MediaScanner.java
```

在本节的内容中，将详细讲解 MediaScanner 系统中 Java 层的具体实现过程。

3.2.1　加载JNI库

在文件 MediaScanner.java 中，首先定义 MediaScanner 类并加载 JNI 库，然后定义 JNI 的 Native(本地)函数。主要代码如下所示：

```
public class MediaScanner
{
```

```
    static {
        System.loadLibrary("media jni");
        native_init();
    }

    private final static String TAG = "MediaScanner";

    private static final String[] FILES_PRESCAN_PROJECTION = new String[] {
        Files.FileColumns._ID, // 0
        Files.FileColumns.DATA, // 1
        Files.FileColumns.FORMAT, // 2
        Files.FileColumns.DATE_MODIFIED, // 3
    };

    private static final String[] ID_PROJECTION = new String[] {
        Files.FileColumns._ID,
    };

    private static final int FILES_PRESCAN_ID_COLUMN_INDEX = 0;
    private static final int FILES_PRESCAN_PATH_COLUMN_INDEX = 1;
    private static final int FILES PRESCAN FORMAT COLUMN INDEX = 2;
    private static final int FILES_PRESCAN_DATE_MODIFIED_COLUMN_INDEX = 3;

    private static final String[] PLAYLIST_MEMBERS_PROJECTION = new String[] {
        Audio.Playlists.Members.PLAYLIST_ID, // 0
    };

    private static final int ID_PLAYLISTS_COLUMN_INDEX = 0;
    private static final int PATH PLAYLISTS COLUMN INDEX = 1;
    private static final int DATE_MODIFIED_PLAYLISTS_COLUMN_INDEX = 2;

    private static final String RINGTONES_DIR = "/ringtones/";
    private static final String NOTIFICATIONS_DIR = "/notifications/";
    private static final String ALARMS_DIR = "/alarms/";
    private static final String MUSIC_DIR = "/music/";
    private static final String PODCAST_DIR = "/podcasts/";
    ...
    private static native final void native_init(); //声明一个 native 函数,
                                                     //native 为关键字
    private native final void native_setup();
    ...
}
```

函数 native_init 位于包 android.media 中，其完整路径名为：

```
android.media.MediaScanner.nantive_init
```

根据规则，其对应的 JNI 层函数名称为：

```
android_media_MediaScanner_native_init
```

在调用函数 native 之前，需要先加载 JNI 库，一般在类的 static 中加载调用函数 System.loadLibrary()。在加载了相应的 JNI 库之后，如果要使用相应的 native 函数，只需使用 native 声明需要被调用的函数即可：

```
private native void processDirectory(String path, String extensions,
  MediaScannerClient client);
private native void processFile(String path, String mimeType,
  MediaScannerClient client);
public native void setLocale(String locale);
```

3.2.2 实现扫描工作

在文件 MediaScanner.java 中，通过函数 scanDirectories 实现扫描工作，具体实现代码如下所示：

```
public void scanDirectories(String[] directories, String volumeName) {
    try {
        long start = System.currentTimeMillis();
        initialize(volumeName); //初始化
        prescan(null, true); //扫描前的预处理
        long prescan = System.currentTimeMillis();

        if (ENABLE_BULK_INSERTS) {
            // create MediaInserter for bulk inserts
            mMediaInserter = new MediaInserter(mMediaProvider, 500);
        }
        //函数processDirectory是一个Native函数，功能是对目标文件夹进行扫描
        for (int i=0; i<directories.length; i++) {
            processDirectory(directories[i], mClient);
        }

        if (ENABLE_BULK_INSERTS) {
            // flush remaining inserts
            mMediaInserter.flushAll();
            mMediaInserter = null;
        }

        long scan = System.currentTimeMillis();
        postscan(directories); //扫描后的处理
        long end = System.currentTimeMillis();

        if (false) {
            Log.d(TAG, " prescan time: " + (prescan - start) + "ms\n");
            Log.d(TAG, "    scan time: " + (scan - prescan) + "ms\n");
            Log.d(TAG, "postscan time: " + (end - scan) + "ms\n");
            Log.d(TAG, "   total time: " + (end - start) + "ms\n");
        }
    } catch (SQLException e) {
```

```
        // this might happen if the SD card is removed
        // while the media scanner is running
        Log.e(TAG, "SQLException in MediaScanner.scan()", e);
    } catch (UnsupportedOperationException e) {
        // this might happen if the SD card is removed
        // while the media scanner is running
        Log.e(TAG, "UnsupportedOperationException in MediaScanner.scan()", e);
    } catch (RemoteException e) {
        Log.e(TAG, "RemoteException in MediaScanner.scan()", e);
    }
}
```

在上述代码中，用到了函数 initialize，此函数的功能是实现初始化操作，具体实现代码如下所示：

```
private void initialize(String volumeName) {
    //打开 MediaProvider，获得它的一个实例
    mMediaProvider = mContext.getContentResolver().acquireProvider("media");
    //得到一些 uri
    mAudioUri = Audio.Media.getContentUri(volumeName);
    mVideoUri = Video.Media.getContentUri(volumeName);
    mImagesUri = Images.Media.getContentUri(volumeName);
    mThumbsUri = Images.Thumbnails.getContentUri(volumeName);
    mFilesUri = Files.getContentUri(volumeName);
    //如果需要外部存储的话，则可以支持播放列表，用缓存池实现，例如 mGenreCache 等
    if (!volumeName.equals("internal")) {
        // we only support playlists on external media
        mProcessPlaylists = true;
        mProcessGenres = true;
        mPlaylistsUri = Playlists.getContentUri(volumeName);
        mCaseInsensitivePaths = true;
    }
}
```

3.2.3　读取并保存信息

在文件 MediaScanner.java 中，函数 prescan 的功能是读取先前扫描的数据库中与文件相关的信息并保存起来。此函数创建了一个 FileCache，用来缓存扫描文件的一些信息，例如 last_modified 等。这个 FileCache 是从 MediaProvider 中的已有信息构建出来的，也就是历史信息。后面根据扫描得到的新信息来对应更新历史信息。函数 prescan 的具体实现代码如下所示：

```
private void prescan(String filePath, boolean prescanFiles)
  throws RemoteException {
    Cursor c = null;
    String where = null;
    String[] selectionArgs = null;
    //mPlayLists 保存从数据库中获取的信息
    if (mPlayLists == null) {
```

```
        mPlayLists = new ArrayList<FileEntry>();
    } else {
        mPlayLists.clear();
    }

    if (filePath != null) {
        //只有一个文件查询
        where = MediaStore.Files.FileColumns._ID + ">?"
          + " AND " + Files.FileColumns.DATA + "=?";
        selectionArgs = new String[] {"", filePath};
    } else {
        where = MediaStore.Files.FileColumns._ID + ">?";
        selectionArgs = new String[] {""};
    }

    //告诉提供者不删除文件.
    //如果不需要删除文件，则需要避免意外删除这个文件的机制
    //这可能在系统未被安装和未安装在扫描仪之前发生
    Uri.Builder builder = mFilesUri.buildUpon();
    builder.appendQueryParameter(MediaStore.PARAM_DELETE_DATA, "false");
    MediaBulkDeleter deleter =
      new MediaBulkDeleter(mMediaProvider, builder.build());

    //根据内容提供者建立文件列表
    try {
        if (prescanFiles) {
            //首先从文件表读到现有文件
            //因为可能存在删除不存在文件的情况，所以要小批量地实现数据库查询以避免这个问题
            long lastId = Long.MIN_VALUE;
            Uri limitUri = mFilesUri.buildUpon()
              .appendQueryParameter("limit", "1000").build();
            mWasEmptyPriorToScan = true;

            while (true) {
                selectionArgs[0] = "" + lastId;
                if (c != null) {
                    c.close();
                    c = null;
                }
                c = mMediaProvider.query(limitUri, FILES_PRESCAN_PROJECTION,
                  where, selectionArgs, MediaStore.Files.FileColumns._ID, null);
                if (c == null) {
                    break;
                }

                int num = c.getCount();

                if (num == 0) {
                    break;
```

```
            }
            mWasEmptyPriorToScan = false;
            while (c.moveToNext()) {
                long rowId = c.getLong(FILES_PRESCAN_ID_COLUMN_INDEX);
                String path = c.getString(FILES_PRESCAN_PATH_COLUMN_INDEX);
                int format = c.getInt(FILES_PRESCAN_FORMAT_COLUMN_INDEX);
                long lastModified =
                  c.getLong(FILES_PRESCAN_DATE_MODIFIED_COLUMN_INDEX);
                lastId = rowId;

                // Only consider entries with absolute path names.
                // This allows storing URIs in the database without the
                // media scanner removing them.
                if (path!=null && path.startsWith("/")) {
                    boolean exists = false;
                    try {
                        exists = Libcore.os.access(
                          path, libcore.io.OsConstants.F_OK);
                    } catch (ErrnoException e1) {}
                    if (!exists && !MtpConstants.isAbstractObject(format)) {
                        // do not delete missing playlists,
                        // since they may have been
                        // modified by the user.
                        // The user can delete them in the media player instead.
                        // instead, clear the path and lastModified fields
                        // in the row
                        MediaFile.MediaFileType mediaFileType =
                          MediaFile.getFileType(path);
                        int fileType = (mediaFileType == null?
                          0 : mediaFileType.fileType);

                        if (!MediaFile.isPlayListFileType(fileType)) {
                            deleter.delete(rowId);
                            if (path.toLowerCase(Locale.US)
                              .endsWith ("/.nomedia")) {
                                deleter.flush();
                                String parent = new File(path).getParent();
                                mMediaProvider.call(
                                  MediaStore.UNHIDE_CALL, parent, null);
                            }
                        }
                    }
                }
            }
        }
    }
}
finally {
```

```
        if (c != null) {
            c.close();
        }
        deleter.flush();
    }

    //计算图像的原始尺寸
    mOriginalCount = 0;
    c = mMediaProvider.query(mImagesUri, ID_PROJECTION, null, null, null, null);
    if (c != null) {
        mOriginalCount = c.getCount();
        c.close();
    }
}
```

3.2.4 删除不是SD卡中的文件信息

在文件 MediaScanner.java 中，函数 postscan 的功能是删除不存在于 SD 卡中的文件信息。函数 postscan 的具体实现代码如下所示：

```
private void postscan(String[] directories) throws RemoteException {

    //触发播放列表后能够知道对应存储的媒体文件
    if (mProcessPlaylists) {
        processPlayLists();
    }

    if (mOriginalCount == 0
      && mImagesUri.equals(Images.Media.getContentUri("external")))
        pruneDeadThumbnailFiles();

    //允许 GC 清理
    mPlayLists = null;
    mMediaProvider = null;
}
```

3.2.5 直接转向JNI

在文件 MediaScanner.java 中，processDirectory 是一个本地方法，能够直接转向 JNI。具体实现代码如下所示：

```
static void android_media_MediaScanner_processDirectory(JNIEnv *env,
  jobject thiz, jstring path, jstring extensions, jobject client)
{   //获取 MediaScanner
    MediaScanner *mp = (MediaScanner*)env->GetIntField(thiz, fields.context);
    //参数判断，并抛出异常
    if (path == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
```

```
        return;
    }
    if (extensions == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }

    const char *pathStr = env->GetStringUTFChars(path, NULL);
    if (pathStr == NULL) {  // Out of memory
        jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
        return;
    }
    const char *extensionsStr = env->GetStringUTFChars(extensions, NULL);
    if (extensionsStr == NULL) {  // Out of memory
        env->ReleaseStringUTFChars(path, pathStr);
        jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
        return;
    }
    //初始化 client 实例
    MyMediaScannerClient myClient(env, client);
    //mp 调用 processDirectory
    mp->processDirectory(pathStr, extensionsStr, myClient, ExceptionCheck, env);
    //gc
    env->ReleaseStringUTFChars(path, pathStr);
    env->ReleaseStringUTFChars(extensions, extensionsStr);
}
```

3.2.6　扫描函数scanFile

在此讲解 Java 层的函数 scanFile，功能是调用函数 doScanFile 对指定的文件进行扫描，具体实现代码如下所示：

```
public void scanFile(String path, long lastModified, long fileSize,
  boolean isDirectory, boolean noMedia) {
    //这是来自本地代码的回调函数
    //Log.v(TAG, "scanFile: " + path);
    doScanFile(path, null, lastModified, fileSize, isDirectory, false, noMedia);
}
```

3.2.7　异常处理

为了处理 Java 实现的方法中或者 C/C++实现的方法中抛出的 Java 异常，JNI 也提供了一套异常处理机制函数集，专门用于检查、分析和处理异常情况。例如在文件 jni.h 中，定义了主要的异常函数，具体代码如下所示：

```
//抛出异常
jint (*Throw)(JNIEnv*, jthrowable);
```

```
//抛出新的异常
jint (*ThrowNew)(JNIEnv*, jclass, const char*);
//异常产生
jthrowable (*ExceptionOccurred)(JNIEnv*);
void (*ExceptionDescribe)(JNIEnv*);
//清除异常
void (*ExceptionClear)(JNIEnv*);
void (*FatalError)(JNIEnv*, const char*);
```

例如，在 Camera 模块中也用到了异常处理，在文件 android_hardware_Camera.cpp 中，也涉及到了异常操作，具体代码如下所示：

```
void JNICameraContext::copyAndPost(JNIEnv *env, const sp<IMemory>& dataPtr,
  int msgType) {
    ...
    if (obj == NULL) {
        LOGE("Couldn't allocate byte array for JPEG data");
        env->ExceptionClear();
    } else {
        env->SetByteArrayRegion(obj, 0, size, data);
    } else {
        LOGE("image heap is NULL");
    }
}
```

在文件 android_hardware_Camera.cpp 中，函数 android_hardware_Camera_startPreview()也同样用到了异常处理机制，具体代码如下所示：

```
static void android_hardware_Camera_startPreview(
  JNIEnv *env, jobject thiz) {
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述代码中，android_hardware_Camera_startPreview()如果发现 startPreview()函数返回错误，则会抛出异常并返回。这里的异常跟 Java 中的异常机制很相似，读者可以对比分析它们的原理。

3.3　分析MediaScanner的JNI层

由于 Android 的应用层的类都是以 Java 写的，这些 Java 类编译为 Dex 形式的 Bytecode 之后，必须借助于 Dalvik 虚拟机(Virtual Machine，VM)来执行并实现。VM 在 Android 系统中扮

演了一个很重要的角色，并且在执行 Java 类的过程中，如果 Java 类需要与 C 组件沟通，VM 就会去载入 C 组件，然后让 Java 的函数顺利地调用到 C 组件的函数。此时，VM 扮演着桥梁的角色，让 Java 与 C 组件能通过标准的 JNI 界面而相互沟通。

应用层的 Java 类是在虚拟机上执行的，而 C 组件不是在 VM 上执行。如果 Java 程序又要求 VM 载入(Load)所指定的 C 组件，可以使用如下所示的指令实现这个功能：

```
System.loadLibrary(*.so 的文件名);
```

例如，在 Android 框架里所提供的 MediaPlayer.java 类中包含了下面的指令：

```
public class MediaPlayer {
    static {
        System.loadLibrary("media_jni");
    }
}
```

这要求 VM 去载入 Android 的/system/lib/libmedia_jni.so 库。载入*.so 后，Java 类与*.so 文件就汇合起来一起执行。

在 JNI 层中，MediaScanner 的对应文件是：

```
./frameworks/base/media/jni/android_media_MediaScanner.cpp
```

在本节的内容中，将详细讲解 MediaScanner 系统中 JNI 层的基本源码。

3.3.1　将Native对象的指针保存到Java对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_init 的功能是将 Native 对象的指针保存到 Java 对象中。函数 android_media_MediaScanner_native_init 的具体实现代码如下所示：

```
static const char* const kClassMediaScanner = "android/media/MediaScanner";
...
/* native_init 函数的 JNI 层实现*/
static void android_media_MediaScanner_native_init(JNIEnv *env) {
    ALOGV("native_init");
    jclass clazz = env->FindClass(kClassMediaScanner);
    if (clazz == NULL) {
        return;
    }
    fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
    if (fields.context == NULL) {
        return;
    }
}
```

3.3.2　创建Native层的MediaScanner对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_setup

的功能是创建一个 Native 层的 MediaScanner 对象，但是此函数使用的是 OpenCore 提供的 PVMediaScanner。

函数 android_media_MediaScanner_native_setup 的具体实现代码如下所示：

```
static void android_media_MediaScanner_native_setup(JNIEnv *env, jobject thiz)
{
    ALOGV("native_setup");
    MediaScanner *mp = new StagefrightMediaScanner;
    if (mp == NULL) {
        jniThrowException(env, kRunTimeException, "Out of memory");
        return;
    }
    env->SetIntField(thiz, fields.context, (int)mp);
}
```

3.4　分析MediaScanner的Native层

Java 的 Native 函数与 JNI 函数是一一对应的关系，在 Android 中使用 JNI NativeMethod 的结构体来记录这种对应关系。在本节的内容中，将详细分析 MediaScanner 系统中的 Native 层的实现源码。

3.4.1　注册JNI函数

在 Android 系统中，使用了一种“特定”的方式来定义其 Native 函数，这与传统定义 Java JNI 的方式有些差别。其中很重要的区别是在 Android 中使用了一种 Java 和 C 函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型是 JNINativeMethod，具体定义如下所示：

```
typedef struct {
    const char *name;          /* Java 中函数的名字 */
    const char *signature;     /* 描述了函数的参数和返回值 */
    void *fnPtr;               /* 函数指针，指向 C 函数 */
} JNINativeMethod;
```

在上述代码中，比较难以理解的是第二个参数，例如：

```
"()V"
"(II)V"
"(Ljava/lang/String;Ljava/lang/String;)V"
```

实际上，这些字符是与函数的参数类型一一对应的，具体说明如下所示：

- “()”中的字符表示参数，后面的则代表返回值。例如“()V”就表示 void Func();。
- “(II)V”表示 void Func(int, int);。

具体的每一个字符的对应关系如下所示：

字符　　　Java 类型　　　C 类型

V	void	void
Z	jboolean	boolean
I	jint	int
J	jlong	long
D	jdouble	double
F	jfloat	float
B	jbyte	byte
C	jchar	char
S	jshort	short

而数组则以“[”开始，用两个字符表示：

字符	Java 类型	C 类型
[I	jintArray	int[]
[F	jfloatArray	float[]
[B	jbyteArray	byte[]
[C	jcharArray	char[]
[S	jshortArray	short[]
[D	jdoubleArray	double[]
[J	jlongArray	long[]
[Z	jbooleanArray	boolean[]

上面的都是基本类型，如果 Java 函数的参数是 class，则以“L”开头，以“;”结尾，中间部分是用“/”隔开的包及类名。而其对应的 C 函数名的参数则为 jobject. 一个例外是 String 类，其对应的类为 jstring，即：

- Ljava/lang/String 中的 String jstring。
- Ljava/net/Socket 中的 Socket jobject。

如果 Java 函数位于一个嵌入类中，则使用$作为类名间的分隔符。例如：

```
(Ljava/lang/String;Landroid/os/FileUtils$FileStatus;)Z"
```

定义并注册 JNINativeMethod 数组，对应的实现代码如下所示：

```
/*定义一个 JNINativeMethod 数组*/
static JNINativeMethod gMethods[] = {
    {
        "processDirectory",
        "(Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
        (void*)android_media_MediaScanner_processDirectory
    },

    {
        "processFile",
        "(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
        (void*)android_media_MediaScanner_processFile
    },
```

```
    {
        "setLocale",
        "(Ljava/lang/String;)V",
        (void*)android_media_MediaScanner_setLocale
    },

    {
        "extractAlbumArt",
        "(Ljava/io/FileDescriptor;)[B",
        (void*)android_media_MediaScanner_extractAlbumArt
    },

    {
        "native_init",
        "()V",
        (void*)android_media_MediaScanner_native_init
    },

    {
        "native_setup",
        "()V",
        (void*)android_media_MediaScanner_native_setup
    },

    {
        "native_finalize",
        "()V",
        (void*)android_media_MediaScanner_native_finalize
    },
};
/*注册JNINativeMethod数组*/
int register_android_media_MediaScanner(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
             kClassMediaScanner, gMethods, NELEM(gMethods));
}
```

3.4.2 完成注册工作

定义并注册数组 JNINativeMethod 后，接着需要调用函数 registerNativeMethods 来完成调用工作。函数 registerNativeMethods 在文件 AndroidRuntime.cpp 中实现，具体实现代码如下所示：

```
int AndroidRuntime::registerNativeMethods(JNIEnv *env,
   const char *className, const JNINativeMethod *gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

在上述代码中，jniRegisterNativeMethods 是 Android 为方便 JNI 使用而提供的一个帮助函数，此函数在文件 JNIHelp.cpp 中实现，具体实现代码如下所示：

```
extern "C" int jniRegisterNativeMethods(C JNIEnv *env, const char *className,
  const JNINativeMethod *gMethods, int numMethods)
{
    JNIEnv *e = reinterpret_cast<JNIEnv*>(env);

    ALOGV("Registering %s natives", className);
    scoped_local_ref<jclass> c(env, findClass(env, className));
    if (c.get() == NULL) {
        ALOGE("Native registration unable to find class '%s', aborting", className);
        abort();
    }
    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) {
        ALOGE("RegisterNatives failed for '%s', aborting", className);
        abort();
    }
    return 0;
}
```

通过上述代码，可以了解函数 registerNativeMethods 的作用。应用层级的 Java 类别通过 VM 调用本地函数，这个过程通常是通过 VM 去寻找“*.so”格式库文件中的本地函数。如果需要连续调用很多次，则需要每次都寻找一遍，这与会多花费很多时间。此时，组件开发人员可以自行向 VM 登记本地函数。例如，在 Android 的/system/lib/libmedia_jni.so 文件里的代码片段如下所示：

```
//#define LOG_NDEBUG 0
#define LOG_TAG "MediaPlayer-JNI"
static JNINativeMethod gMethods[] = {
    {"setDataSource",  "(Ljava/lang/String;)V",
     (void*)android_media_MediaPlayer_setDataSource},
    {"setDataSource",  "(Ljava/io/FileDescriptor;JJ)V",
     (void*)android_media_MediaPlayer_setDataSourceFD},

    {"prepare",   "()V",     (void*)android_media_MediaPlayer_prepare},
    {"prepareAsync",   "()V",   (void*)android_media_MediaPlayer_prepareAsync},
    {" start",   "()V",   (void*)android media MediaPlayer start},
    {"_stop",   "()V",    (void*)android_media_MediaPlayer_stop},
    {"getVideoWidth",   "()I",     (void*)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight",   "()I",     (void*)android_media_MediaPlayer_getVideoHeight},
    {"seekTo",          "(I)V",      (void*)android_media_MediaPlayer_seekTo},
    {"_pause",          "()V",       (void*)android_media_MediaPlayer_pause},
    {"isPlaying",         "()Z",       (void*)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition", "()I", (void*)android_media_MediaPlayer_getCurrentPosition},
    {"getDuration",      "()I",      (void*)android_media_MediaPlayer_getDuration},
    {"_release",          "()V",       (void*)android_media_MediaPlayer_release},
    {"_reset",            "()V",         (void*)android_media_MediaPlayer_reset},
    {"setAudioStreamType","(I)V",  (void*)android_media_MediaPlayer_setAudioStreamType},
```

```
    {"setLooping",     "(Z)V",     (void*)android_media_MediaPlayer_setLooping},
    {"setVolume",      "(FF)V",    (void*)android_media_MediaPlayer_setVolume},
    {"getFrameAt",     "(I)Landroid/graphics/Bitmap;",
        (void*)android_media_MediaPlayer_getFrameAt},
    {"native_setup",    "(Ljava/lang/Object;)V",
         (void*)android_media_MediaPlayer_native_setup},
    {"native_finalize",    "()V",    (void*)android_media_MediaPlayer_native_finalize},
};
static int register_android_media_MediaPlayer(JNIEnv *env) {
    return AndroidRuntime::registerNativeMethods(env,
              "android/media/MediaPlayer", gMethods, NELEM(gMethods));
}
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
    if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
}
```

这样，VM 载入 libmedia_jni.so 文件时就会调用 JNI_OnLoad()，然后 JNI_OnLoad()调用 register_android_media_MediaPlayer()。此时，就调用 AndroidRuntime::registerNativeMethods()，并向 VM(即 AndroidRuntime)登记数组 gMethods[]表格所含的本地函数。由此可见，函数 registerNativeMethods 具备如下所示的两个功能：

- 更有效率地找到函数。
- 可以在执行期间进行抽换。因为 gMethods[]是一个“<名称，函数指针>”格式的对照表，所以在执行程序时，可以通过多次调用 registerNativeMethods()函数的方式来更换本地函数的指针。

3.4.3 动态注册

当 Java 层通过 System.loadLibrary 加载完 JNI 动态库后，接着会查找函数 JNI_OnLoad，通过调用 JNI_OnLoad 函数完成动态注册工作。

函数 JNI_OnLoad 在文件 android_media_MediaPlayer.cpp 中实现，具体代码如下所示：

```
jint JNI_OnLoad(JavaVM *vm, void *reserved)
{
    JNIEnv *env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**)&env, JNI_VERSION_1_4) != JNI_OK) {
        ALOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    ...
    if (register_android_media_MediaScanner(env) < 0) {
        ALOGE("ERROR: MediaScanner native registration failed\n");
```

```
        goto bail;
    }
    ...
    /*成功——则返回有效的版本号*/
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

函数 JNI_OnLoad 会回传 JNI_VERSION_1_4 的值给 VM，这使 VM 能够知道所使用 JNI 的版本是什么。此外，它也做一些初期的动作(可调用任何本地函数)，例如下面的指令：

```
if (register_android_media_MediaPlayer(env) < 0) {
    LOGE("ERROR: MediaPlayer native registration failed\n");
    goto bail;
}
```

这样，就将此组件提供的各个本地函数(Native Function)登记到 VM 里，以便能加快后续调用本地函数的效率。

函数 JNI_OnUnload()与 JNI_OnLoad()是相对的。载入 C 组件时，会立即调用 JNI_OnLoad()进行组件内的初期动作；而当 VM 释放该 C 组件时，则会调用 JNI_OnUnload()函数来进行善后清除动作。当 VM 调用 JNI_OnLoad()或 JNI_Unload()函数时，都会将 VM 的指针(Pointer)传递给它们，其参数如下所示：

```
jint JNI_OnLoad(JavaVM *vm, void *reserved) {}
jint JNI_OnUnload(JavaVM *vm, void *reserved) {}
```

在 JNI_OnLoad()函数中，通过 VM 的指针而取得 JNIEnv 的指针值，并存入 env 指针变数中，如下述指令：

```
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
    JNIEnv *env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**)&env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
}
```

由于 VM 通常是多线程(Multi-threading)的执行环境。每一个线程在调用 JNI_OnLoad()时，所传递进来的 JNIEnv 指针都是不同的。为了配合这种多线程的环境，C 组件开发者在撰写本地函数时，可通过 JNIEnv 指针的不同而避免线程数据冲突问题，以确保所写的本地函数能安全地在 Android 的多线程 VM 里安全地执行。基于这个理由，当调用 C 组件的函数时，都会将 JNIEnv 指针传递给它，对应的代码如下所示：

```
jint JNI OnLoad(JavaVM *vm, void *reserved) {
    JNIEnv *env = NULL;
    if (register_android_media_MediaPlayer(env) < 0) { }
}
```

这样，当 JNI_OnLoad()调用函数 register_android_media_MediaPlayer(env)时，就将 env 指针传递过去。这样函数 register_android_media_MediaPlayer()就能借用该标识值来区别不同的线程，以便解决数据冲突的问题。

例如，在 register_android_media_MediaPlayer()函数中，可以编写如下所示的指令：

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
}
```

此时可以查看是否有其他线程程序进入此对象，如果没有，则此线程就进入该对象中执行了。并且也可以编写如下所示的指令：

```
if ((*env)->MonitorExit(env, obj) != JNI_OK) {
}
```

这样便可以查看是否此线程正在此对象内执行，如果是，此线程就会立即离开。

3.4.4　处理路径参数

在文件 frameworks/base/media/libmedia/MediaScanner.cpp 中，函数 processDirectory 的功能是对路径参数进行一些处理，调用 doProcessDirectory。里面的参数@extensions 可能包含多个扩展名，在扩展名之间用“,”分隔开。具体实现代码如下所示：

```
status_t MediaScanner::processDirectory(const char *path, const char *extensions,
  MediaScannerClient &client, ExceptionCheck exceptionCheck,
  void *exceptionEnv) {
    int pathLength = strlen(path);
    if (pathLength >= PATH_MAX) {
        return UNKNOWN_ERROR;
    }
    char *pathBuffer = (char*)malloc(PATH_MAX + 1);
    if (!pathBuffer) {
        return UNKNOWN_ERROR;
    }
    int pathRemaining = PATH_MAX - pathLength;
    strcpy(pathBuffer, path);
    if (pathLength>0 && pathBuffer[pathLength-1]!='/') {
        pathBuffer[pathLength] = '/';
        pathBuffer[pathLength + 1] = 0;
        --pathRemaining;
    }
    client.setLocale(locale());
    status_t result = doProcessDirectory(pathBuffer, pathRemaining, extensions,
      client, exceptionCheck, exceptionEnv);

    free(pathBuffer);

    return result;
}
```

3.4.5　扫描文件

当收到扫描某个文件的请求时，会调用函数 scanFile 来扫描这个文件。函数 scanFile 的具体实现代码如下所示：

```
virtual bool scanFile(const char *path, long long lastModified,
  long long fileSize) {
    jstring pathStr;
    if ((pathStr=mEnv->NewStringUTF(path)) == NULL) return false;
    //调用 Java 中 mClient 的 scanFile 方法
    mEnv->CallVoidMethod(
      mClient, mScanFileMethodID, pathStr, lastModified, fileSize);
    mEnv->DeleteLocalRef(pathStr);
    return (!mEnv->ExceptionCheck());
}
```

3.4.6　添加TAG信息

在文件\frameworks\av\media\libmedia\MediaScannerClient.cpp 中，通过函数 addStringTag 添加 TAG 信息。这个 MediaScannerClient 是在 opencore 的 MediaScanner.cpp 文件中实现的，而文件 android_media_MediaScanner.cpp 中的 MyMediaScannerClient 是从 MediaScannerClient 派生下来的。

函数 addStringTag 的具体实现代码如下所示：

```
status_t MediaScannerClient::addStringTag(
  const char *name, const char *value) {

    if (mLocaleEncoding != kEncodingNone) {

        //不要缓存，都是 ASCII 字符串。
        //调用 handlestringtag 直接代替。
        //查看值中是否有非 ASCII 字符，应该是 UTF8)
        bool nonAscii = false;
        const char *chp = value;
        char ch;
        while ((ch = *chp++)) {
            if (ch & 0x80) {
                nonAscii = true;
                break;
            }
        }

        //判断 name 和 value 的编码是不是 ASCII，不是则保存到 mNames 和 mValues 中
        // save the strings for later so they can be used for native encoding detection
        mNames->push_back(name);
        mValues->push_back(value);
```

```
        return OK;
    }

    //其他的失败情形

    //如果字符编码是 ASCII，则调用函数 handleStringTag
    return handleStringTag(name, value);
}
```

Java 与 JNI 基本数据类型的转换关系如表 3-1 所示。

表 3-1　基本数据类型的转换关系

Java	Native 类型	本地 C 类型	字　长
boolean	jboolean	无符号	8 位
byte	jbyte	无符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jint	有符号	32 位
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

数组数据类型的对应关系如表 3-2 所示。

表 3-2　数组数据类型的对应关系

字　符	Java 类型	C 类型
[I	jintArray	int[]
[F	jfloatArray	float[]
[B	jbyteArray	byte[]
[C	jshortArray	short[]
[D	jdoubleArrary	double[]
[J	jlongArray	long[]
[S	jshortArray	short[]
[Z	jbooleanArray	booean[]

对象数据类型的对应关系如表 3-3 所示。

表 3-3　对象数据类型的对应关系

对　象	Java 类型	C 类型
Ljava/lang/String	String	jstring
Ljava/net/Socket	Socket	jobject

引用数据类型的转换关系如图 3-3 所示。

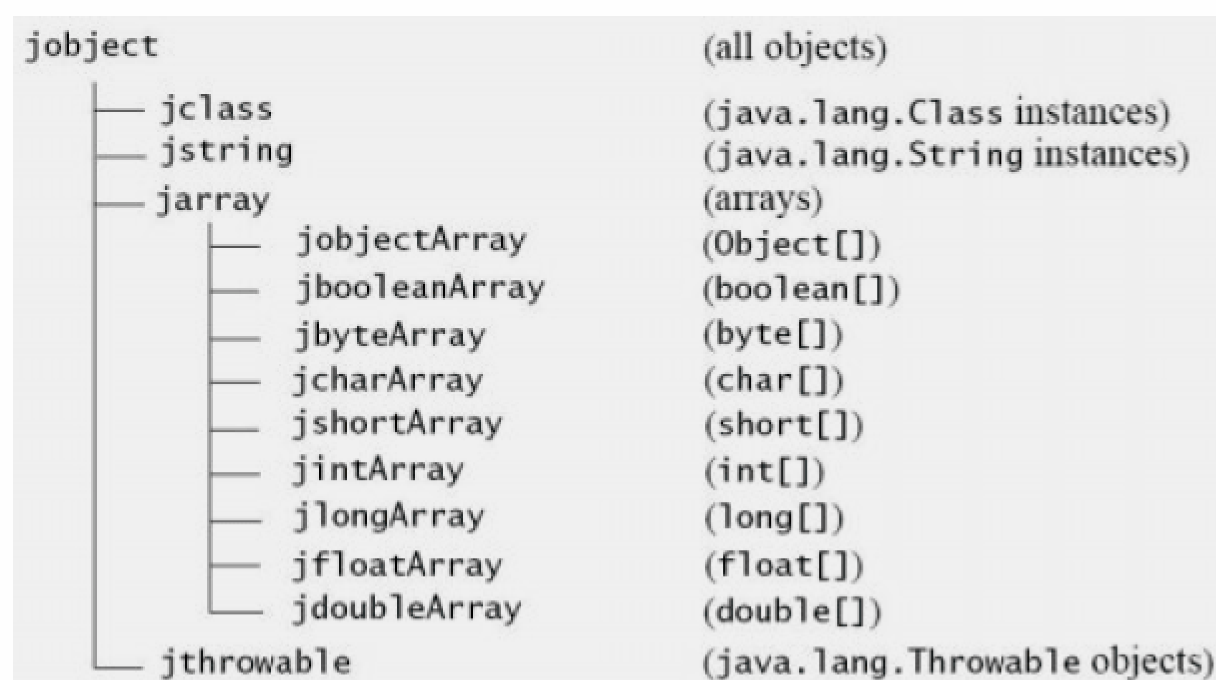


图 3-3　引用数据类型的转换关系

3.4.7　JNIEnv接口

JNIEnv 是跟线程相关的，在 Native Method(本地方法)中，JNIEnv 作为第一个参数传入。JNIEnv 的内部结构如图 3-4 所示。

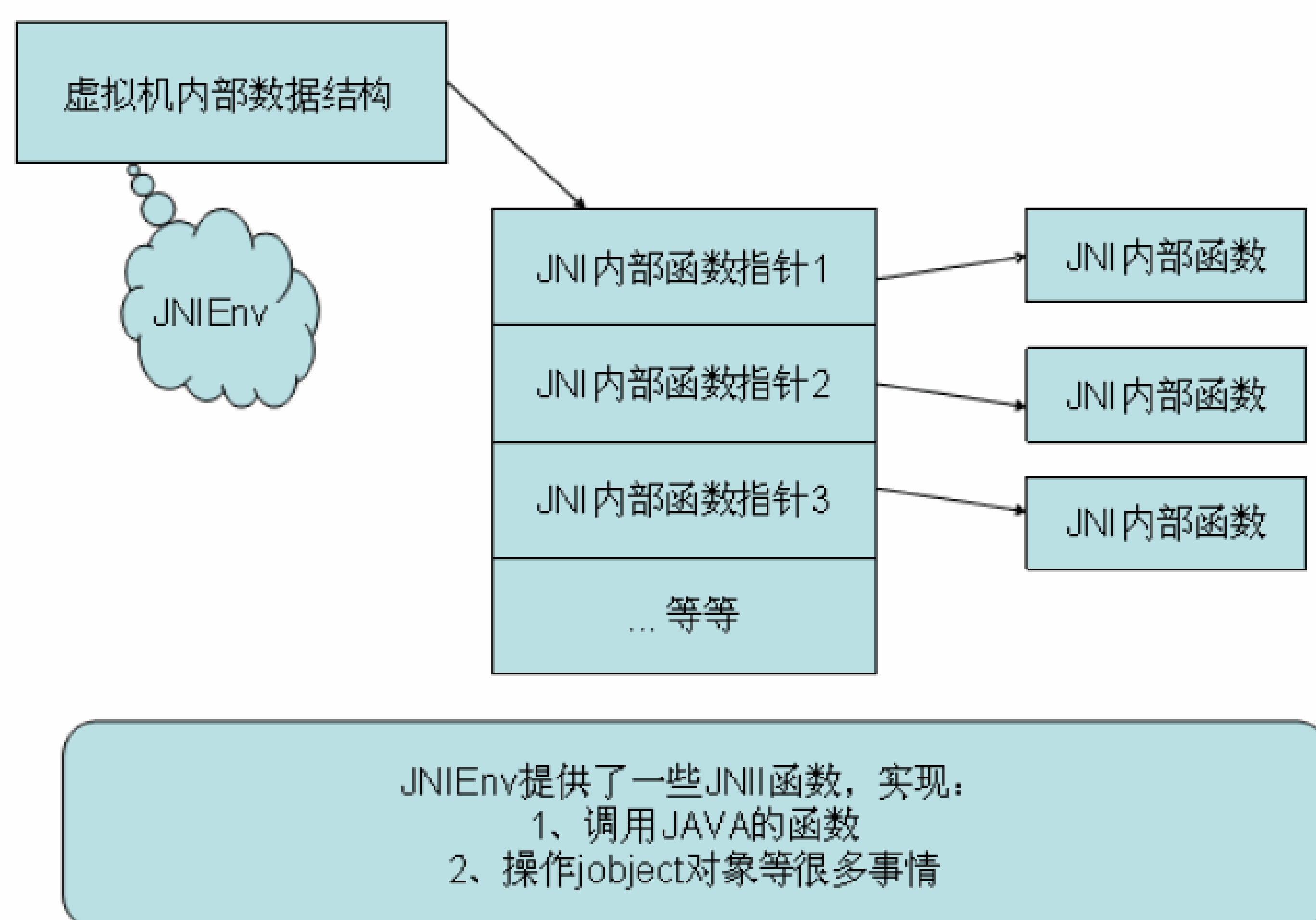


图 3-4　JNIEnv的内部结构

在 JNIEnv 不作为参数传入的时候，JNI 提供了如下所示的两个函数获得它：

```
(*jvm)->AttachCurrentThread(jvm, (void**)&env, NULL)
(*jvm)->GetEnv(jvm, (void**)&env, JNI_VERSION_1_2)
```

上述两个函数都利用 JavaVM 接口获得 JNIEnv 接口，并且 JNI 可以将获得的 JNIEnv 封装成一个函数。即：

```
JNIEnv* JNU_GetEnv() {
    JNIEnv *env;
    (*g_jvm)->GetEnv(g_jvm, (void**)&env, JNI_VERSION_1_2);
    return env;
}
```

Java 通过 JNI 机制调用 C/C++写的 native 程序，C/C++开发的 Native 程序需要遵循一定的 JNI 规范。例如，下面的例子就是一个 JNI 函数声明：

```
JNIEXPORT jint JNICALL Java jnitest MyTest test(
  JNIEnv *env, jobject obj, jint arg0);
```

JVM 负责从 Java Stack 转入 C/C++ Native Stack。当 Java 进入 JNI 调用时，除了函数本身的参数(arg0)外，会多出两个参数：JNIEnv 指针和 jobject 指针。其中 JNIEnv 指针是 JVM 创建的，用于 Native 的 C/C++方法操纵 Java 执行栈中的数据，比如 Java Class、Java Method 等。

首先，JNI 对于 JNIEnv 的使用提供了两种语法，分别是 C 语法以及 C++语法。

其中 C 语法是：

```
jsize len = (*env)->GetArrayLength(env, array);
```

C++语法是：

```
jsize len = env->GetArrayLength(array);
```

因为 C 语言并不支持对象的概念，所以 C 语法中需要把 env 作为第一个参数传入，这类似于 C++的隐式参数 this 指针。

另外，在使用 JNIEnv 接口时，需要遵循如下所示的两个设计原则。

(1) JNIEnv 指针被设计成了 Thread Local Storage(TLS)变量，也就是说，针对每一个 Thread，JNIEnv 变量都有独立的 Copy。不能把 Thead#1 使用的 JNIEnv 传给 Thread#2 使用。

(2) 在 JNIEnv 中定义了一组函数指针，C/C++ Native 程序是通过这些函数指针操纵 Java 数据的。这样设计的好处是，C/C++程序不需要依赖任何函数库或者 DLL。由于 JVM 可能由不同的厂商实现，不同厂商有自己不同的 JNI 实现，如果要求这些厂商暴露约定好的一些头文件和库，这不是灵活的设计。而且使用函数指针表的另外一个好处是，JVM 可以根据启动参数动态替换 JNI 实现。

在 jint JNI_OnLoad(JavaVM *vm, void *reserved)的整个进程中只有一个 JavaVM 对象，可以保存并在任何地方使用没有问题。利用 JavaVM 中的 AttachCurrentThread 函数，就可以得到这个线程的 JNIEnv 结构体，利用 DetachCurnetThread 释放相应的资源。

3.4.8 JNI中的环境变量

在 Android 系统中的所有模块的 JNI 层的代码中，经常看到函数中有 JNIEnv*类型的参数，例如文件/frameworks/base/core/jni/android_hardware_Camera.c 中的如下代码：

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz) {
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述函数中，第一个参数为 JNIEnv *Env，此处的 JNIEnv*类型是一个指向 JNI 环境的指针，JNIEnv*类型在文件 Jni.h 中定义，在此结构体中包含了一些 JNI 中常用到的函数和一组函数指针，C/C++正是通过这些函数指针来操作 Java 函数的。JNIEnv 结构体在文件 jni.h 中定义，具体实现代码如下所示：

```
struct _JNIEnv {
    ...
    jint GetVersion()
    { return functions->GetVersion(this); }
    ...
    jclass FindClass(const char *name)
    { return functions->FindClass(this, name); }
    ...
    void CallVoidMethodA(jobject obj, jmethodID methodID, jvalue *args)
    { functions->CallVoidMethodA(this, obj, methodID, args); }
    ...
    jmethodID GetStaticMethodID(jclass clazz, const char *name, const char *sig)
    { return functions->GetStaticMethodID(this, clazz, name, sig); }
    ...
}
```

通过上述代码可以发现，正是通过 JNIEnv 这个指针，我们才能够调用 JNI 环境中的一些方法。

3.5　JNI实例分析(基于Camera系统)

在本节的内容中，将以 Camera 系统中的预览功能作为素材，在 Android 4.3 源码中详细分析 JNI 机制衔接 Java 层和 C/C++层的过程，并分析 Java 层调用底层代码来实现预览功能的具体方法。

3.5.1　Java层预览接口

在本小节的主要内容中，将详细介绍 Camera 模块中预览功能的 Java 层的文件路径，以及其中预览函数的功能和作用。Camera 中的 Java 层代码在 Camera.java 文件中实现，其详细路径为：

```
/Package/apps/camera/src/com/android/camemra/Camera.java
```

在文件 Camera.java 中定义了预览相关的函数 startPreview()和 stopPreview()，这是图像预览的入口函数。如下代码详细介绍了文件 Camera.java 中 startPreview()和 stopPreview()这两个函数的功能：

```
//开始预览
private void startPreview() {
    if (mPausing || isFinishing()) return;
    mFocusManager.resetTouchFocus();
```

```
    mCameraDevice.setErrorCallback(mErrorCallback);
    // If we're previewing already, stop the preview first (this will blank
    // the screen).
    if (mCameraState != PREVIEW_STOPPED) stopPreview();
    setPreviewDisplay(mSurfaceHolder);
    setDisplayOrientation();
    if (!mSnapshotOnIdle) {
        // If the focus mode is continuous autofocus, call cancelAutoFocus to
        // resume it because it may have been paused by autoFocus call.
        if (Parameters.FOCUS_MODE_CONTINUOUS_PICTURE
          .equals (mFocusManager.getFocusMode())) {
            mCameraDevice.cancelAutoFocus();
        }
        mFocusManager.setAeAwbLock(false); // Unlock AE and AWB.
    }
    //设置Camera的参数
    setCameraParameters(UPDATE_PARAM_ALL);
    // Inform the mainthread to go on the UI initialization.
    if (mCameraPreviewThread != null) {
        synchronized (mCameraPreviewThread) {
            mCameraPreviewThread.notify();
        }
    }
    try {
        Log.v(TAG, "startPreview");
        //调用框架层的Camera类来实现预览功能
        mCameraDevice.startPreview();
    } catch (Throwable ex) {
        closeCamera();
        throw new RuntimeException("startPreview failed", ex);
    }
    mZoomState = ZOOM_STOPPED;
    setCameraState(IDLE);
    mFocusManager.onPreviewStarted();
    if (mSnapshotOnIdle) {
        mHandler.post(mDoSnapRunnable);
    }
}
//停止预览
private void stopPreview() {
    //判断Camera的状态
    if (mCameraDevice!=null && mCameraState!=PREVIEW_STOPPED) {
        Log.v(TAG, "stopPreview");
        mCameraDevice.cancelAutoFocus(); // Reset the focus.
        mCameraDevice.stopPreview();
        mFaceDetectionStarted = false;
    }
    //设置Camera的状态
    setCameraState(PREVIEW_STOPPED);
```

```
    mFocusManager.onPreviewStopped();
}
```

上述代码演示了 Java 层的函数功能，在 Android 的框架层封装了 Camera 的框架层类 Camera.java，文件路径为：

```
frameworks/base/code/java/android/hardware
```

在类 Camera 中声明了很多 native 的方法，比如 startPreview()、stopPreview()，具体声明代码如下所示：

```
public native final void startPreview();
public native final void stopPreview();
```

上述声明的函数 native 会直接注册到 JNI 中，然后调用 C/C++层的 startPreview()和 stopPrevire()函数。

文件在 android_hardware_Camera.cpp 中完成注册 Camera 预览函数的功能，具体文件路径为：

```
frameworks/base/core/jni/android_hardware_Camera.cpp
```

3.5.2　注册预览的JNI函数

在本小节的主要内容中，将详细介绍将 Camera 模块的预览功注册到 JNI 系统中的方法。在文件 android_hardware_Camera.cpp 中，会将 Camera 模块中的所有接口函数注册到 JNI 系统中，文件 android_hardware_Camera.cpp 中的具体注册代码如下：

```
//初始化 JNI 中的 Java 对象并且注册 Camera 模块的 JNI 函数
int register_android_hardware_Camera(JNIEnv *env) {
   field fields_to_find[] = {
      { "android/hardware/Camera", "mNativeContext",   "I", &fields.context },
      { "android/view/Surface",  ANDROID_VIEW_SURFACE_JNI_ID,
        "I", &fields.surface },
      { "android/graphics/SurfaceTexture",
        ANDROID_GRAPHICS_SURFACETEXTURE_JNI_ID, "I", &fields.surfaceTexture },
      { "android/hardware/Camera$CameraInfo", "facing",   "I",
         &fields.facing },
      { "android/hardware/Camera$CameraInfo", "orientation",   "I",
         &fields.orientation },
      { "android/hardware/Camera$Face", "rect", "Landroid/graphics/Rect;",
         &fields.face_rect },
      { "android/hardware/Camera$Face", "score", "I", &fields.face_score },
      { "android/graphics/Rect", "left", "I", &fields.rect_left },
      { "android/graphics/Rect", "top", "I", &fields.rect_top },
      { "android/graphics/Rect", "right", "I", &fields.rect_right },
      { "android/graphics/Rect", "bottom", "I", &fields.rect_bottom },
   };
   if (find_fields(env, fields_to_find, NELEM(fields_to_find)) < 0)
      return -1;
```

```
    jclass clazz = env->FindClass("android/hardware/Camera");
    fields.post event = env->GetStaticMethodID(clazz, "postEventFromNative",
      "(Ljava/lang/Object;IIILjava/lang/Object;)V");
    if (fields.post_event == NULL) {
        LOGE("Can't find android/hardware/Camera.postEventFromNative");
        return -1;
    }
    clazz = env->FindClass("android/graphics/Rect");
    fields.rect_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.rect_constructor == NULL) {
        LOGE("Can't find android/graphics/Rect.Rect()");
        return -1;
    }
    clazz = env->FindClass("android/hardware/Camera$Face");
    fields.face_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.face_constructor == NULL) {
        LOGE("Can't find android/hardware/Camera$Face.Face()");
        return -1;
    }
    // Register native functions
    // 注册接口函数到 JNI 中
    return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
                                                 camMethods, NELEM(camMethods));
```

在上述代码中，函数 register_android_hardware_Camera()的功能是初始化 Java 的 Camera 相关的类对象，并且将接口函数注册到 JNI 中，在文件 android_hardware_Camera.cpp 中，Camera 的函数映射表如下所示：

```
static JNINativeMethod camMethods[] = {
  { "getNumberOfCameras",
    "()I",
    (void *)android_hardware_Camera_getNumberOfCameras },
  { "getCameraInfo",
    "(ILandroid/hardware/Camera$CameraInfo;)V",
    (void*)android_hardware_Camera_getCameraInfo },
  { "native setup",
    "(Ljava/lang/Object;I)V",
    (void*)android_hardware_Camera_native_setup },
  { "native_release",
    "()V",
    (void*)android_hardware_Camera_release },
  { "setPreviewDisplay",
    "(Landroid/view/Surface;)V",
    (void*)android_hardware_Camera_setPreviewDisplay },
  { "setPreviewTexture",
    "(Landroid/graphics/SurfaceTexture;)V",
    (void*)android_hardware_Camera_setPreviewTexture },
  //开始预览
  { "startPreview",
```

```
  "()V",
  (void*)android_hardware_Camera_startPreview },
//停止预览
{ "_stopPreview",
  "()V",
  (void*)android_hardware_Camera_stopPreview },
{ "previewEnabled",
  "()Z",
  (void*)android_hardware_Camera_previewEnabled },
{ "setHasPreviewCallback",
  "(ZZ)V",
  (void*)android_hardware_Camera_setHasPreviewCallback },
{ "_addCallbackBuffer",
  "([BI)V",
  (void*)android_hardware_Camera_addCallbackBuffer },
{ "native_autoFocus",
  "()V",
  (void*)android_hardware_Camera_autoFocus },
{ "native_cancelAutoFocus",
  "()V",
  (void*)android_hardware_Camera_cancelAutoFocus },
{ "native_takePicture",
  "(I)V",
  (void*)android_hardware_Camera_takePicture },
{ "native_setParameters",
  "(Ljava/lang/String;)V",
  (void*)android_hardware_Camera_setParameters },
{ "native_getParameters",
  "()Ljava/lang/String;",
  (void*)android_hardware_Camera_getParameters },
{ "reconnect",
  "()V",
  (void*)android_hardware_Camera_reconnect },
{ "lock",
  "()V",
  (void*)android_hardware_Camera_lock },
{ "unlock",
  "()V",
  (void*)android_hardware_Camera_unlock },
{ "startSmoothZoom",
  "(I)V",
  (void*)android_hardware_Camera_startSmoothZoom },
{ "stopSmoothZoom",
  "()V",
  (void*)android_hardware_Camera_stopSmoothZoom },
{ "setDisplayOrientation",
  "(I)V",
  (void*)android_hardware_Camera_setDisplayOrientation },
{ "_startFaceDetection",
```

```
    "(I)V",
    (void*)android_hardware_Camera_startFaceDetection },
  { "_stopFaceDetection",
    "()V",
    (void*)android_hardware_Camera_stopFaceDetection},
};
```

根据上述代码可以看到，有了这个函数映射表，则 Camera 的 Java 层接口可以调用到 C/C++层的接口函数，C/C++层的预览函数指针名为 android_hardware_Camera_startPreview()、android_hardware_Camera_stopPreview()，这两个函数会进而调用到 C/C++层的函数，在文件 android_hardware_Camera.cpp 中。

具体代码如下所示：

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz) {
    ALOGV("startPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    //调用 C/C++层的 startPreview()
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
static void android_hardware_Camera_stopPreview(JNIEnv *env, jobject thiz) {
    ALOGV("stopPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> c = get_native_camera(env, thiz, NULL);
    if (c == 0) return;
    //调用 C/C++层的 stopPreview()
    c->stopPreview();
}
```

3.5.3 C/C++层的预览函数

Camera 模块的 C/C++层文件路径为/frameworks/av/camera/Camera.cpp，具体的实现代码如下所示：

```
//开始预览
status_t Camera::startPreview() {
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    //调用其他 so 库的 startPreview()
    return c->startPreview();
}
//停止预览
void Camera::stopPreview() {
```

```
    ALOGV("stopPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return;
    //调用其他 so 库的 stopPreview()
    c->stopPreview();
}
```

通过上述代码发现，文件 Camere.cpp 的功能是实现 Camera 模块中的 C/C++层，然后继续调用更加底层的预览的实现代码。

第 4 章
Android内存系统分析

内存(Memory)也被称为内存储器，其作用是暂时存放 CPU 中的运算数据以及与硬盘等外部存储器交换的数据。

内存是当今计算机中的重要部件之一，是用户任务与 CPU 处理器进行沟通的桥梁。只要是运行中的计算机，CPU 就会把需要运算的数据调到内存中进行运算处理，当运算完成后，CPU 再将结果传送出来。

其实，智能手机就是一台微型的 PC，也具有与计算机一样的结构，例如 CPU 和内存。本章将和读者一起探讨 Android 4.3 中内存系统的基本知识，为步入本书后面知识的学习打下基础。

4.1 Android的进程通信机制

要想实现对 Android 系统内存的优化，需要首先了解 Android 的内存系统，了解内存控制进程运行的机制。在本节的内容中，将带领读者一起探讨和分析 Android 的进程通信机制。

4.1.1 Android的进程间通信(IPC)机制Binder

在 Android 系统中，每一个应用程序都是由一些 Activity 和 Service 组成的，一般 Service 运行在独立的进程中，而 Activity 可能运行在同一个进程中，也有可能运行在不同的进程中。那么，不在同一个进程的 Activity 或者 Service 之间究竟是如何通信的呢？下面将介绍的 Binder 进程间通信机制就是用来实现这个功能的。

众所周知，Android 系统是基于 Linux 内核的，而 Linux 内核继承和兼容了丰富的 Unix 系统进程间通信(IPC)机制。有传统的管道(Pipe)、信号(Signal)和跟踪(Trace)，这三项通信手段只能用于父进程和子进程之间，或者只用于兄弟进程之间。随着技术的发展，后来又增加了命令管道(Named Pipe)，这样，使得进程之间的通信不再局限于父子进程或者兄弟进程之间。为了更好地支持商业应用中的事务处理，在 AT&T 的 Unix 系统 V 中，又增加了如下三种称为“System V IPC”的进程间通信机制：

- 报文队列(Message)。
- 共享内存(Share Memory)。
- 信号量(Semaphore)。

Android 系统没有采用上述提到的各种进程间通信机制，而是采用 Binder 机制。其实 Binder 并不是 Android 提出来的一套新的进程间通信机制，它是基于 OpenBinder 来实现的。Binder 是一种进程间通信机制，这是一种类似于 COM 和 CORBA 的分布式组件架构。通俗地说，其实是提供远程过程调用(RPC)功能。从英文字面上的意思看，Binder 具有粘结剂的意思，那么它把什么东西粘结在一起呢？在 Android 系统中，Binder 机制由一些系统组件组成：Client、Server、Service Manager 和 Binder 驱动程序，其中 Client、Server 和 Service Manager 运行在用户空间，Binder 驱动程序运行在内核空间。Binder 就是一种把这 4 个组件粘合在一起的粘结剂了。其中的核心组件便是 Binder 驱动程序，Service Manager 提供了辅助管理的功能，Client 和 Server 正是在 Binder 驱动和 Service Manager 提供的基础设施上，实现 Client/Server 之间的通信。Service Manager 和 Binder 驱动已经在 Android 平台中实现完毕，开发者只要按照规范实现自己的 Client 和 Server 组件即可。对于初学者来说，Android 系统的 Binder 机制是最难理解的了，而 Binder 机制无论从系统开发还是应用开发的角度来看，都是 Android 系统中最重要的组成部分，所以很有必要深入了解 Binder 的工作方式。要深入了解 Binder 的工作方式，最好的方式是阅读 Binder 相关的源代码了，因为 Linux 的鼻祖 Linus Torvalds 曾经说过一句名言：Read The Fucking Source Code。

要想深入理解 Binder 机制，必须了解 Binder 在用户空间的三个组件 Client、Server 和 Service Manager 之间的相互关系，并了解内核空间中 Binder 驱动程序的数据结构和设计原理。具体来说，Android 系统 Binder 机制中的 4 个组件 Client、Server、Service Manager 和 Binder 驱动程

序的相互关系如图 4-1 所示。

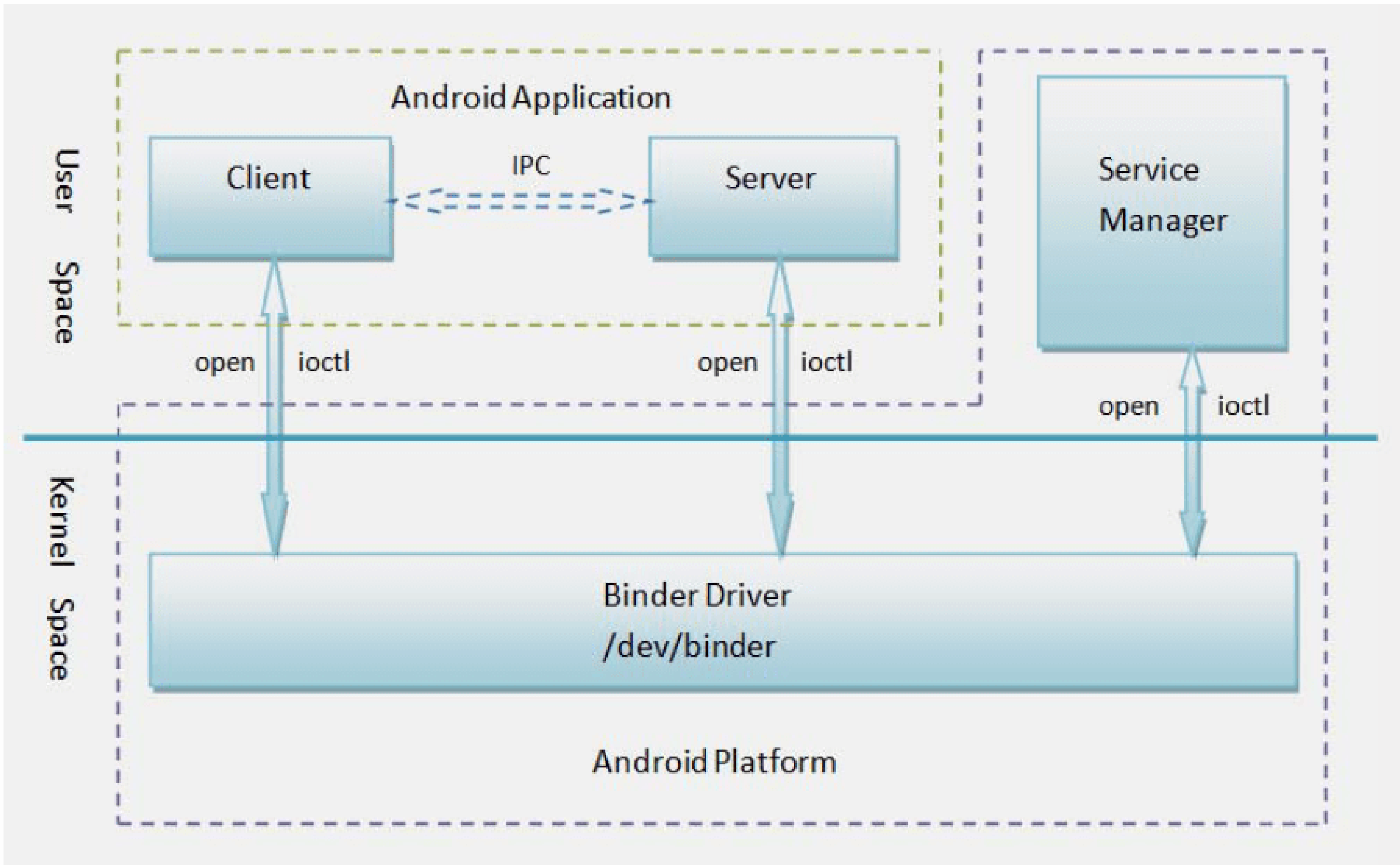


图 4-1　组件Client、Server、Service Manager和Binder驱动程序的相互关系

图 4-1 表示的关系的具体说明如下。

(1) Client、Server 和 Service Manager 实现在用户空间中，Binder 驱动程序实现在内核空间中。

(2) Binder 驱动程序和 Service Manager 在 Android 平台中已经实现，开发者只需要在用户空间实现自己的 Client 和 Server 即可。

(3) Binder 驱动程序提供设备文件/dev/binder 与用户空间交互，Client、Server 和 Service Manager 通过文件操作函数 open()和 ioctl()与 Binder 驱动程序进行通信。

(4) Client 和 Server 之间的进程间通信通过 Binder 驱动程序间接实现。

(5) Service Manager 是一个保护进程，用来管理 Server，并向 Client 提供查询 Server 接口的能力。

4.1.2　Service Manager是Binder机制的上下文管理者

在分析 Binder 源代码时，需要先弄清楚 Service Manager 告知 Binder 驱动程序自己是 Binder 机制的上下文管理者的过程。Service Manager 是整个 Binder 机制的保护进程，用来管理开发者创建的各种 Server，并且向 Client 提供查询 Server 远程接口的功能。

因为 Service Manager 组件是用来管理 Server 并且向 Client 提供查询 Server 远程接口的功能的，所以 Service Manager 必然要与 Server 以及 Client 进行通信。Service Manger、Client 和 Server 三者分别是运行在独立的进程中的，这样它们之间的通信也属于进程间的通信，而且也是采用 Binder 机制进行进程间通信。因此，Service Manager 在充当 Binder 机制的保护进程的角色的同时，也在充当 Server 的角色，也是一种特殊的 Server。

Service Manager 在用户空间的源代码位于 frameworks/base/cmds/servicemanager 目录中，

主要由文件 binder.h、binder.c 和 service_manager.c 组成。Service Manager 在 Binder 机制中的基本执行流程如图 4-2 所示。

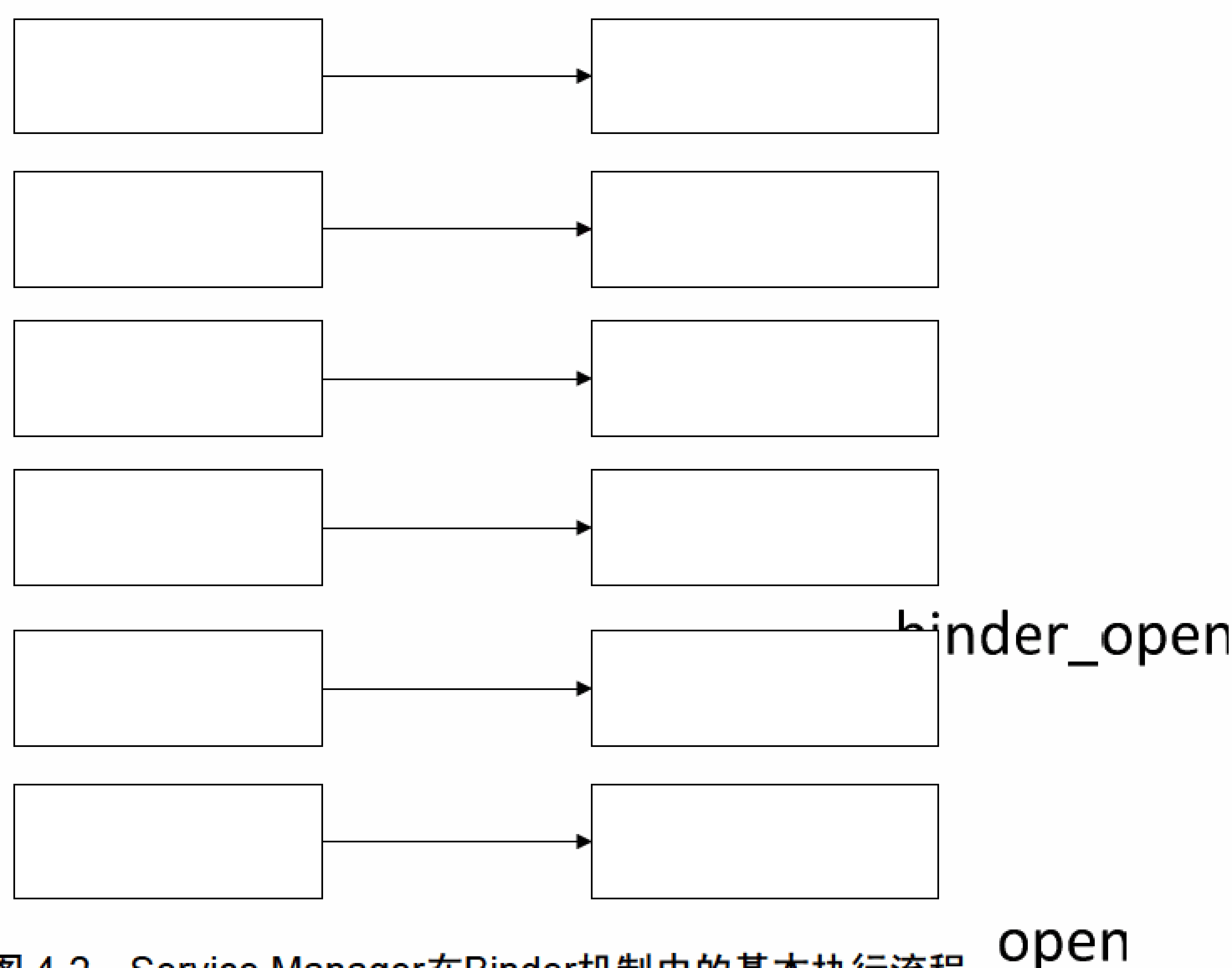


图 4-2 Service Manager在Binder机制中的基本执行流程

4.2 分析Ashmem驱动程序

在 Android 系统中提供了独特的匿名共享内存子系统 Ashmem，全称是 Anonymous Shared Memory。Ashmem 以驱动程序的形式在内核空间中实现，具有如下所示的两个特点：

- 能够辅助内存管理系统来有效地管理不再使用的内存块。
- 通过 Binder 进程间通信机制来实现进程间的内存共享。

对于 Android 系统的匿名共享内存子系统来说，其主体是以驱动程序的形式实现在内核空间的，同时，在系统运行时，库层和应用程序框架层提供了访问接口。其中，在系统运行时库层提供了 C/C++调用接口，而在应用程序框架层提供了 Java 调用接口。在此，我们将直接通过应用程序框架层提供的 Java 调用接口来说明匿名共享内存子系统 Ashmem 的使用方法，毕竟我们在 Android 开发应用程序时，是基于 Java 语言的。其实应用程序框架层的 Java 调用接口是通过 JNI 方法来调用系统运行时库层的 C/C++调用接口，最后进入到内核空间的 Ashmem 驱动程序的。

在 Android 4.3 系统中，匿名共享内存 Ashmem 驱动程序利用了 Linux 的共享内存子系统导出的接口来实现自己的功能。Android 匿名共享内存系统的核心功能是实现创建(open)、映射(mmap)、读写(read/write)以及锁定和解锁(pin/unpin)，接下来将一一讲解上述内容。

4.2.1 基础数据结构

在 Ashmem 驱动程序中，用到了 ashmem_area、ashmem_range 和 ashmem_range 三个结构

体。其中前两个结构体在文件 kernel/goldfish/mm/ashmem.c 中定义，实现代码如下所示：

```
struct ashmem_area {
    char name[ASHMEM_FULL_NAME_LEN]; /* 匿名共享内存的名称 */
    struct list_head unpinned_list;  /* 解锁内存列表 */
    struct file *file;              /* 指向临时文件系统 tmpfs 中的一个文件 */
    size_t size;                    /* 文件大小 */
    unsigned long prot_mask;         /* 匿名共享内存的访问保护位 */
};
struct ashmem_range {
    struct list_head lru;           /* 最近最少使用的列表 */
    struct list_head unpinned;      /* entry in its area's unpinned list */
    struct ashmem_area *asma;       /* associated area */
    size_t pgstart;                /* 处于解锁状态内存的开始地址 */
    size_t pgend;                  /* 处于解锁状态内存的结束地址 */
    unsigned int purged;           /* 解锁内存是否被收回 */
};
```

结构体 ashmem_area 用于表示一块匿名共享内存单元，结构体 ashmem_range 用于表示处于解锁状态的内存。

结构体 ashmem_range 用于表示被锁定或被解锁的内存，在文件 kernel/goldfish/include/linux/ashmem.h 中定义，具体代码如下所示：

```
struct ashmem_pin {
    __u32 offset;   /* 这块内存的偏移值 */
    __u32 len;      /* 这块内存的大小 */
};
```

结构体 ashmem_fops 定义了 dev/ashmem 的操作方法列表，具体代码如下所示：

```
static struct file_operations ashmem_fops = {
    .owner = THIS_MODULE,
    .open = ashmem_open,
    .release = ashmem_release,
    .mmap = ashmem_mmap,
    .unlocked_ioctl = ashmem_ioctl,
    .compat_ioctl = ashmem_ioctl,
};
```

4.2.2　初始化处理

通过 Ashmem 驱动初始化函数，可以获取如下所示的两点信息：

- Ashmem 给用户空间暴露了什么接口，即创建了什么样的设备文件。
- Ashmem 提供了什么函数来操作这个设备文件。

Ashmem 驱动程序在文件 kernel/common/mm/ashmem.c 中实现，其中函数 ashmem_init 实现模块初始化处理，主要实现代码如下所示：

```
static struct miscdevice ashmem_misc = {
```

```
    .minor = MISC_DYNAMIC_MINOR,
    .name = "ashmem",
    .fops = &ashmem_fops,
};
static int __init ashmem_init(void)
{
    int ret;
    ...
    ret = misc_register(&ashmem_misc);
    if (unlikely(ret)) {
        printk(KERN_ERR "ashmem: failed to register misc device!\n");
        return ret;
    }
    ...
    return 0;
}
```

在上述代码中，在加载 Ashmem 驱动程序时，会创建一个设备文件/dev/ashmem，这是一个 misc 类型的设备。通过函数 misc_register 来注册 misc 设备，调用这个函数后会在/dev 目录下生成一个 ashmem 设备文件。在设备文件中一共提供了 open、mmap、release 和 ioctl 四种操作，此处并没有 read 和 write 操作，原因是读写共享内存的方法是通过内存映射地址来进行的，通过 mmap 系统调用将这个设备文件映射到进程地址空间中。与此同时，直接对内存进行了读写操作，所以不需要通过 read 和 write 方式进行文件操作。

匿名共享内存创建功能是在文件 frameworks/base/core/java/android/os/MemoryFile.java 中实现的，此文件调用了 MemoryFile 类的构造函数，MemoryFile 的构造函数调用了 JNI 函数 native_open，这样便创建了匿名内存共享文件。JNI 方法 native_open 在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示：

```
static jobject android_os_MemoryFile_open(JNIEnv *env, jobject clazz,
 jstring name, jint length) {
    const char *namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);

    int result = ashmem_create_region(namestr, length);

    if (name)
        env->ReleaseStringUTFChars(name, namestr);

    if (result < 0) {
        jniThrowException(
          env, "java/io/IOException", "ashmem_create_region failed");
        return NULL;
    }

    return jniCreateFileDescriptor(env, result);
}
```

函数 native_open 通过运行时库提供的接口 ashmem_create_region 创建匿名共享内存，这个

接口在文件 system/core/libcutils/ashmem-dev.c 中实现，具体代码如下所示：

```
int ashmem_create_region(const char *name, size_t size) {
    int fd, ret;

    fd = open(ASHMEM_DEVICE, O_RDWR);
    if (fd < 0)
        return fd;

    if (name) {
        char buf[ASHMEM_NAME_LEN];

        strlcpy(buf, name, sizeof(buf));
        ret = ioctl(fd, ASHMEM_SET_NAME, buf);
        if (ret < 0)
            goto error;
    }

    ret = ioctl(fd, ASHMEM_SET_SIZE, size);
    if (ret < 0)
        goto error;

    return fd;

error:
    close(fd);
    return ret;
}
```

在上述代码中，通过执行三个文件操作系统调用的方式与 Ashmem 驱动程序进行交互。通过 open 操作打开设备文件 ASHMEM_DEVICE，通过 ioctl 操作设置匿名共享内存的名称和大小。

4.2.3　打开匿名共享内存设备文件

open 进入内核后，会调用函数 ashmem_open 打开匿名共享内存设备文件，此函数能够为程序创建一个 ashmem_area 结构体，具体实现代码如下所示：

```
static int ashmem_open(struct inode *inode, struct file *file) {
    struct ashmem_area *asma;
    int ret;
    ret = nonseekable_open(inode, file);
    if (unlikely(ret))
        return ret;
    asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
    if (unlikely(!asma))
        return -ENOMEM;
    INIT_LIST_HEAD(&asma->unpinned_list);
    memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
```

```
    asma->prot_mask = PROT_MASK;
    file->private data = asma;
    return 0;
}
```

上述代码的执行流程如下所示。

(1) 通过函数 nonseekable_open 设置这个文件不可以执行定位操作，即不可执行 seek 文件操作。

(2) 通过函数 kmem_cache_zalloc 在刚创建的 slab 缓冲区 ashmem_area_cachep 中创建一个 ashmem_area 结构体，并将创建的结构体保存在本地变量 asma 中。

(3) 初始化变量 asma 的其他域，其中域 name 初始为宏 ASHMEM_NAME_PREFIX，宏 ASHMEM_NAME_PREFIX 的定义代码为：

```
#define ASHMEM NAME PREFIX "dev/ashmem/"
#define ASHMEM_NAME_PREFIX_LEN (sizeof(ASHMEM_NAME_PREFIX) - 1)
```

(4) 将结构 ashmem_area 保存在打开文件结构体的 private_data 域中，此时通过使用 Ashmem 驱动程序，可以在其他模块通过 private_data 域来取回这个 ashmem_area 结构。

在函数 ashmem_create_region 中调用了两次 ioctl 文件操作，功能是设置新建匿名共享内存的名字和大小。在文件 kernel/comon/mm/include/ashmem.h 中，ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 分别表示新建内存的名字和大小，具体定义代码如下所示：

```
#define ASHMEM_NAME_LEN      256
#define __ASHMEMIOC      0x77
#define ASHMEM_SET_NAME      _IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
#define ASHMEM_SET_SIZE      _IOW(__ASHMEMIOC, 3, size_t)
```

其中 ASHMEM_SET_NAME 的 ioctl 调用会进入到 Ashmem 驱动程序函数 ashmem_ioctl 中，此函数能够将从用户空间传进来的匿名共享内存的大小值保存在对应的 asma->size 域中。函数 ashmem_ioctl 的实现代码如下所示：

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct ashmem_area *asma = file->private_data;
    long ret = -ENOTTY;
    switch (cmd) {
        case ASHMEM_SET_NAME:
            ret = set_name(asma, (void __user*)arg);
            break;
        case ASHMEM_GET_NAME:
            ret = get_name(asma, (void __user*)arg);
            break;
        case ASHMEM_SET_SIZE:
            ret = -EINVAL;
            if (!asma->file) {
                ret = 0;
                asma->size = (size t)arg;
            }
```

```
            break;
        case ASHMEM_GET_SIZE:
            ret = asma->size;
            break;
        case ASHMEM_SET_PROT_MASK:
            ret = set_prot_mask(asma, arg);
            break;
        case ASHMEM_GET_PROT_MASK:
            ret = asma->prot_mask;
            break;
        case ASHMEM_PIN:
        case ASHMEM_UNPIN:
        case ASHMEM_GET_PIN_STATUS:
            ret = ashmem_pin_unpin(asma, cmd, (void __user*)arg);
            break;
        case ASHMEM_PURGE_ALL_CACHES:
            ret = -EPERM;
            if (capable(CAP_SYS_ADMIN)) {
                ret = ashmem_shrink(0, GFP_KERNEL);
                ashmem_shrink(ret, GFP_KERNEL);
            }
            break;
    }
    return ret;
}
```

上述代码主要完成如下两个功能。

- struct ashmem_area *asma = file->private_data：获取描述将要改名的匿名共享内存 asma。
- ret = set_name(asma, (void __user*)arg)：调用函数 set_name 修改匿名共享内存的名称。

函数 set_name 也是在文件 kernel/goldfish/mm/ashmem.c 中实现的，功能是把用户空间传进来的匿名共享内存的名字设置到 asma->name 域中。函数 set_name 的具体实现代码如下所示：

```
static int set_name(struct ashmem_area *asma, void __user *name)
{
    int ret = 0;
    mutex_lock(&ashmem_mutex);
    /* cannot change an existing mapping's name */
    if (unlikely(asma->file)) {
        ret = -EINVAL;
        goto out;
    }
    if (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
      name, ASHMEM_NAME_LEN)))
        ret = -EFAULT;
    asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
out:
    mutex_unlock(&ashmem_mutex);
```

```
    return ret;
}
```

到此为止，创建匿名共享内存的过程就全部介绍完毕了。

4.2.4　内存映射

Ashmem 驱动程序并不提供文件的 read 操作和 write 操作，如果进程要访问这个共享内存，则必须将这个设备文件映射到自己的进程空间中，然后才能进行内存访问。在类 MemoryFile 的构造函数中，创建匿名共享内存后，需要把匿名共享内存设备文件映射到进程空间。映射功能是通过调用 JNI 方法 native_mmap 实现的，此 JNI 方法在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示：

```
static jint android_os_MemoryFile_mmap(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在上述代码中，从匿名设备文件/dev/ashmem 获得文件描述符 fd。有个这个文件描述符后，就可以直接通过函数 mmap 执行内存映射操作了。当调用函数 mmap 打开映射到进程的地址空间时，会立即执行 ashmem 去的中的函数 ashmem_mmap。函数 ashmem_mmap 的功能是，调用 Linux 内核中的函数 shmem_file_setup 在临时文件系统 tmpfs 中创建一个临时文件，这个临时文件与 Ashmem 驱动程序创建的匿名共享内存对应。函数 ashmem_mmap 在文件 kernel/goldfish/mm/ashmem.c 中定义，具体实现代码如下所示：

```
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
    struct ashmem_area *asma = file->private_data;
    int ret = 0;
    mutex_lock(&ashmem_mutex);
    /* user needs to SET_SIZE before mapping */
    if (unlikely(!asma->size)) {
        ret = -EINVAL;
        goto out;
    }
    /* requested protection bits must match our allowed protection mask */
    if (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
        ret = -EPERM;
        goto out;
    }
    if (!asma->file) {
        char *name = ASHMEM_NAME_DEF;
        struct file *vmfile;
```

```
        if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
            name = asma->name;
        /* ... and allocate the backing ashmem file */
        vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
        if (unlikely(IS_ERR(vmfile))) {
            ret = PTR_ERR(vmfile);
            goto out;
        }
        asma->file = vmfile;
    }
    get_file(asma->file);
    if (vma->vm_flags & VM_SHARED)
        shmem_set_file(vma, asma->file);
    else {
        if (vma->vm_file)
            fput(vma->vm_file);
        vma->vm_file = asma->file;
    }
    vma->vm_flags |= VM_CAN_NONLINEAR;
out:
    mutex_unlock(&ashmem_mutex);
    return ret;
}
```

在上述代码中，检查了虚拟内存 vma 是否允许在不同进程之间实现共享。如果允许，则调用函数 shmem_set_file 来设置它的映射文件和内存操作方法表。

4.2.5 读写操作

在 Android 4.3 的源码中，从类 MemoryFile 中可以获得读写操作的过程，对应的代码如下所示：

```
private static native int native_read(FileDescriptor fd, int address,
  byte[] buffer, int srcOffset, int destOffset, int count,
  boolean isUnpinned) throws IOException;
private static native void native_write(FileDescriptor fd, int address,
  byte[] buffer, int srcOffset, int destOffset, int count,
  boolean isUnpinned) throws IOException;
private FileDescriptor mFD;       // ashmem file descriptor
private int mAddress;   // address of ashmem memory
private int mLength;    // total length of our ashmem region
private boolean mAllowPurging = false;  // true if our ashmem region is unpinned
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
  throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't read from deactivated memory file.");
    }
    if (destOffset<0 || destOffset>buffer.length || count<0
      || count>buffer.length-destOffset
```

```
      || srcOffset<0 || srcOffset>mLength
      || count>mLength-srcOffset) {
        throw new IndexOutOfBoundsException();
    }
    return native_read(
      mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
  throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't write to deactivated memory file.");
    }
    if (srcOffset<0 || srcOffset>buffer.length || count<0
      || count>buffer.length-srcOffset
      || destOffset<0 || destOffset>mLength
      || count>mLength-destOffset) {
        throw new IndexOutOfBoundsException();
    }
    native_write(
      mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

通过对上述代码的分析可知，是通过调用 JNI 方法实现读写匿名共享内存操作功能的。读操作的 JNI 方法是 native_read，写操作的 JNI 方法是 native_write，这两个方法都在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中定义，具体实现代码如下所示：

```
static jint android_os_MemoryFile_read(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset,
  jint destOffset, jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0)==ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->SetByteArrayRegion(buffer, destOffset, count,
      (const jbyte*)address + srcOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}

static jint android_os_MemoryFile_write(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset,
  jint destOffset, jint count, jboolean unpinned)
{
```

```
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0)==ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(buffer, srcOffset, count,
      (jbyte*)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

在上述代码中，函数 ashmem_pin_region 和函数 ashmem_unpin_region 用于为系统运行时库提供接口，功能是执行匿名共享内存的锁定和解锁操作。这样便能够通知 Ashmem 驱动程序哪些内存块是正在使用的，哪些需要锁定，哪些不需要使用，哪些可以解锁。这两个函数在文件 system/core/libcutils/ashmem-dev.c 中定义，具体实现代码如下所示：

```
int ashmem_pin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_PIN, &pin);
}
int ashmem_unpin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_UNPIN, &pin);
}
```

经过上述操作之后，Ashmem 驱动程序就可以在整个内存管理系统中管理内存了。

4.2.6　锁定和解锁

在 Android 系统中，通过如下两个 ioctl 操作实现匿名共享内存的锁定和解锁操作：

- ASHMEM_PIN。
- ASHMEM_UNPIN。

ASHMEM_PIN 和 ASHMEM_UNPIN 在文件 kernel/common/include/linux/ashmem.h 中定义，对应的代码如下所示：

```
#define __ASHMEMIOC      0x77
#define ASHMEM_PIN      _IOW(__ASHMEMIOC, 7, struct ashmem_pin)
#define ASHMEM_UNPIN    _IOW(__ASHMEMIOC, 8, struct ashmem_pin)
struct ashmem_pin {
    __u32 offset;   /* offset into region, in bytes, page-aligned */
    __u32 len;  /* length forward from offset, in bytes, page-aligned */
};
```

再看函数 ashmem_ioctl，在实现代码中与 ASHMEM_PIN 和 ASHMEM_UNPIN 这两个操作相关的代码如下所示：

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct ashmem_area *asma = file->private_data;
    long ret = -ENOTTY;
    switch (cmd) {
    ...
    case ASHMEM_PIN:
    case ASHMEM_UNPIN:
        ret = ashmem_pin_unpin(asma, cmd, (void __user*)arg);
        break;
    ...
    }
    return ret;
}
```

在上述代码中，调用函数 ashmem_pin_unpin 处理控制命令 ASHMEM_PIN 和 ASHMEM_UNPIN。函数 ashmem_pin_unpin 的实现流程如下所示。

① 获取传递到用户空间的参数，并将获取值保存在本地变量 pin 中。这是一个 struct ashmem_pin 类型的结构体类型，在里面包括了要 pin/unpin 的内存块的起始地址和大小。

② 因为起始地址和大小的单位都是字节，所以转换处理为以页面为单位的、并保存在本地变量 pgstart 和 pgend 中。

③ 不但对参数进行安全性检查，并且确保只要从用户空间传进来的内块块的大小值为 0，就认为是要 pin/unpin 整个匿名共享内存。

④ 判断当前要执行操作的类别，根据 ASHMEM_PIN 操作和 ASHMEM_UNPIN 操作分别执行 ashmem_pin 和 ashmem_unpin。

⑤ 当创建匿名共享内存时，所有默认的内存都是 pinned 状态的，只有用户告诉 Ashmem 驱动程序要 unpin 某一块内存时，Ashmem 驱动程序才会把这块内存 unpin。

⑥ 用户告知 Ashmem 驱动程序重新 pin 某一块前面被 unpin 过的内块，这样能够将此内存从 unpinned 状态转换到 pinned 状态。

函数 ashmem_pin_unpin 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,
  void __user *p)
{
    struct ashmem_pin pin;
    size_t pgstart, pgend;
    int ret = -EINVAL;

    if (unlikely(!asma->file))
        return -EINVAL;

    if (unlikely(copy_from_user(&pin, p, sizeof(pin))))
        return -EFAULT;
```

```
    /* per custom, you can pass zero for len to mean "everything onward" */
    if (!pin.len)
        pin.len = PAGE_ALIGN(asma->size) - pin.offset;

    if (unlikely((pin.offset | pin.len) & ~PAGE_MASK))
        return -EINVAL;

    if (unlikely(((__u32) -1) - pin.offset < pin.len))
        return -EINVAL;

    if (unlikely(PAGE_ALIGN(asma->size) < pin.offset + pin.len))
        return -EINVAL;

    pgstart = pin.offset / PAGE_SIZE;
    pgend = pgstart + (pin.len / PAGE_SIZE) - 1;

    mutex_lock(&ashmem_mutex);

    switch (cmd) {
    case ASHMEM_PIN:
        ret = ashmem_pin(asma, pgstart, pgend);
        break;
    case ASHMEM_UNPIN:
        ret = ashmem_unpin(asma, pgstart, pgend);
        break;
    case ASHMEM_GET_PIN_STATUS:
        ret = ashmem_get_pin_status(asma, pgstart, pgend);
        break;
    }

    mutex_unlock(&ashmem_mutex);

    return ret;
}
```

由此可见，执行 ASHMEM_PIN 操作的目标对象必须是一块处于 unpinned 状态的内存块。

函数 ashmem_unpin 的功能是解锁某一块匿名共享内存，具体处理流程如下所示。

① 在遍历 asma->unpinned_list 列表时，查找当前处于 unpinned 状态的内存块是否与将要 unpin 的内存块[pgstart, pgend]相交，如果相交，则通过执行合并操作来调整 pgstart 和 pgend 的大小。

② 调用函数 range_del 删除原来的已经被 unpinned 过的内存块。

③ 调用函数 range_alloc 重新 unpinned 调整过后的内存块[pgstart, pgend]，此时新的内存块[pgstart, pgend]已经包含了刚才所有被删掉的 unpinned 状态的内存。

④ 如果找到相交的内存块，并且调整了 pgstart 和 pgend 的大小之后，需要重新扫描 asma->unpinned_list 列表。原因是新的内存块[pgstart, pgend]可能与前后的处于 unpinned 状态的内存块发生相交。

函数 ashmem_unpin 在文件 kernel/goldfish/ashmem.c 中定义，具体的实现代码如下所示：

```
static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
    struct ashmem_range *range, *next;
    unsigned int purged = ASHMEM_NOT_PURGED;

restart:
    list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
        /* short circuit: this is our insertion point */
        if (range_before_page(range, pgstart))
            break;

        /*
         * The user can ask us to unpin pages that are already entirely
         * or partially pinned. We handle those two cases here.
         */
        if (page_range_subsumed_by_range(range, pgstart, pgend))
            return 0;
        if (page_range_in_range(range, pgstart, pgend)) {
            pgstart = min_t(size_t, range->pgstart, pgstart),
            pgend = max_t(size_t, range->pgend, pgend);
            purged |= range->purged;
            range_del(range);
            goto restart;
        }
    }
    return range_alloc(asma, range, purged, pgstart, pgend);
}
```

range_before_page 的操作是一个宏定义，功能是判断 range 描述的内存块是否在 page 页面之前，如果是，则表示结束整个描述。asma->unpinned_list 列表是按照页面号从大到小进行排列的，并且每一块被 unpin 的内存都是不相交的。range_before_pag 的定义代码如下所示：

```
#define range_before_page(range, page) \
  ((range)->pgend < (page))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块是不是包含了[start, end]这个内存块，如果包含，则说明当前要 unpin 的内存块已经处于 unpinned 状态。如果什么也不用操作，则直接返回。page_range_subsumed_by_range 的定义代码如下所示：

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

page_range_in_range 的操作也是一个宏定义，功能是判断内存块[start, end]是否互相包含或者相交。page_range_in_range 的定义代码如下所示：

```
#define page_range_in_range(range, start, end) \
  (page_in_range(range, start) || page_in_range(range, end) || \
   page_range_subsumes_range(range, start, end))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块 range 是否包含内存块[start, end]。page_range_subsumed_by_range 的定义代码如下所示：

```
#define page range subsumed by range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

range_in_range 的操作也是一个宏定义，功能是判断内存块地址 page 是否包含在内存块 range 中。range_in_range 的定义代码如下所示：

```
#define page_in_range(range, page) \
 (((range)->pgstart <= (page)) && ((range)->pgend >= (page)))
```

再看函数 range_del，功能是从 asma->unpinned_list 中删掉内存块，并判断它是否在 lru 列表中。函数 range_del 的具体实现代码如下所示：

```
static void range_del(struct ashmem_range *range)
{
     list del(&range->unpinned);
     if (range_on_lru(range))
         lru_del(range);
     kmem_cache_free(ashmem_range_cachep, range);
}
```

再看函数 lru_del，内存块的状态 purged 值为 ASHMEM_NOT_PURGED，表示现在没有收回对应的物理页面，那么内存块就位于 lru 列表中，则使用函数 lru_del 删除这个内存块。函数 lru_del 的具体实现代码如下所示：

```
static inline void lru_del(struct ashmem_range *range)
{
     list_del(&range->lru);
     lru_count -= range_size(range);
}
```

再看函数 ashmem_unpin 中调用的 range_alloc 函数，其功能是是从 slab 缓冲区 ashmem_range_cachep 中分配一个 ashmem_range，并进行相应的初始化处理。然后放在对应的列表 ashmem_area->unpinned_list 中，并判断这个 range 的 purged 是否处于 ASHMEM_NOT_PURGED 状态，如果是，则要把它放在 lru 列表中。

函数 range_alloc 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static int range alloc(struct ashmem area *asma,
  struct ashmem_range *prev_range, unsigned int purged,
  size_t start, size_t end)
{
    struct ashmem_range *range;
    range = kmem_cache_zalloc(ashmem_range_cachep, GFP_KERNEL);
    if (unlikely(!range))
       return -ENOMEM;
    range->asma = asma;
    range->pgstart = start;
    range->pgend = end;
```

```
    range->purged = purged;
    list add tail(&range->unpinned, &prev range->unpinned);
    if (range_on_lru(range))
        lru_add(range);
    return 0;
}
```

再看函数 lru_add，功能是将未被回收的已解锁内存块添加到全局列表 ashmem_lru_list 中。函数 lru_add 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static inline void lru_add(struct ashmem_range *range)
{
    list_add_tail(&range->lru, &ashmem_lru_list);
    lru_count += range_size(range);
}
```

再看函数 ashmem_pin，功能是锁定一块匿名共享内存区域。被 pin 的内存块肯定被保存在 unpinned_list 列表中，如果不在，则什么都不用做。要想判断在 unpinned_list 列表中是否存在 pin 的内存块，需要通过遍历 asma->unpinned_list 列表的方式找出与之相交的内存块。函数 ashmem_pin 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
    struct ashmem range *range, *next;
    int ret = ASHMEM_NOT_PURGED;

    list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
        /* moved past last applicable page; we can short circuit */
        if (range_before_page(range, pgstart))
            break;
        if (page_range_in_range(range, pgstart, pgend)) {
            ret |= range->purged;

            /* Case #1: Easy. Just nuke the whole thing. */
            if (page_range_subsumes_range(range, pgstart, pgend)) {
                range_del(range);
                continue;
            }

            /* Case #2: We overlap from the start, so adjust it */
            if (range->pgstart >= pgstart) {
                range_shrink(range, pgend + 1, range->pgend);
                continue;
            }

            /* Case #3: We overlap from the rear, so adjust it */
            if (range->pgend <= pgend) {
                range_shrink(range, range->pgstart, pgstart-1);
                continue;
```

```
            }

            /*
             * Case #4: We eat a chunk out of the middle. A bit
             * more complicated, we allocate a new range for the
             * second half and adjust the first chunk's endpoint.
             */
            range_alloc(asma, range, range->purged,
                    pgend + 1, range->pgend);
            range_shrink(range, range->pgstart, pgstart - 1);
            break;
        }
    }
    return ret;
}
```

在上述代码中，对重新锁定内存块操作实现了判断，通过 if 语句处理了如下所示的 4 种情形：

- 指定要锁定的内存块[start, end]包含了解锁状态的内存块 range，此时只要将解锁状态的内存块 range 从其宿主匿名共享内存的解锁内存块列表 unpinned_list 中删除即可。
- 合并要锁定内存块[pgstart, pgend]的后部和解锁状态内存块 range 的前半部分，此时将解锁状态内存块 range 的开始地址设置为要锁定内存块的末尾地址的下一个页面地址。
- 合并要锁定内存块[pgstart, pgend]的前部和解锁状态内存块 range 的后半部分，此时将解锁状态内存块 range 的末尾地址设置为要锁定内存块的开始地址的下一个页面地址。
- 设置要锁定内存块[pgstart, pgend]包含在解锁状态内存块 range 中。

再看函数 range_shrink，功能是设置 range 描述的内存块的起始页面号，如果还存在于 lru 列表中，则需要调整在 lru 列表中的总页面数大小。函数 range_shrink 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static inline void range_shrink(struct ashmem_range *range,
  size_t start, size_t end)
{
    size_t pre = range_size(range);

    range->pgstart = start;
    range->pgend = end;

    if (range_on_lru(range))
        lru_count -= pre - range_size(range);
}
```

4.2.7　回收内存块

接下来开始看最后一步：回收匿名共享内存块，先回到前面介绍的初始化步骤，分析 Ashmem 驱动初始化函数 ashmem_init，此函数会调用函数 register_shrinker 向内存管理系统注

册一个内存回收算法函数，具体实现代码如下所示：

```
static struct shrinker ashmem_shrinker = {
    .shrink = ashmem_shrink,
    .seeks = DEFAULT_SEEKS * 4,
};

static int __init ashmem_init(void)
{
    ...
    register_shrinker(&ashmem_shrinker);
    printk(KERN_INFO "ashmem: initialized\n");
    return 0;
}
```

其实在 Linux 内核程序中，当系统内存不够用时，内存管理系统就会通过调用内存回收算法的方式删除最近没有用过的内存，将这些内存从物理内存中清除，这样可以增加物理内存的容量。所以在 Android 系统中也借用了这种机制，当内存管理系统回收内存时，会调用函数 ashmem_shrink 以执行内存回收操作。函数 ashmem_shrink 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示：

```
static int ashmem_shrink(struct shrinker *s, struct shrink_control *sc)
{
    struct ashmem_range *range, *next;

    /* We might recurse into filesystem code, so bail out if necessary */
    if (sc->nr_to_scan && !(sc->gfp_mask & __GFP_FS))
        return -1;
    if (!sc->nr_to_scan)
        return lru_count;

    mutex_lock(&ashmem_mutex);
    list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
        loff_t start = range->pgstart * PAGE_SIZE;
        loff_t end = (range->pgend + 1) * PAGE_SIZE;
        do_fallocate(range->asma->file,
                FALLOC_FL_PUNCH_HOLE | FALLOC_FL_KEEP_SIZE,
                start, end - start);
        range->purged = ASHMEM_WAS_PURGED;
        lru_del(range);
        sc->nr_to_scan -= range_size(range);
        if (sc->nr_to_scan <= 0)
            break;
    }

    mutex_unlock(&ashmem_mutex);
    return lru_count;
}
```

4.3　分析C++访问接口层

如果想在 Android 进程之间共享一个完整的匿名共享内存块，可以通过调用接口 MemoryHeapBase 来实现。如果只是想在进程之间共享匿名共享内存块中的一部分，可以通过调用接口 MemoryBase 来实现。

4.3.1　接口MemoryHeapBase

接口 MemoryBase 是以接口 MemoryHeapBase 为基础的，这两个接口都可以作为一个 Binder 对象在进程之间进行传输。因为接口 MemoryHeapBase 是一个 Binder 对象，所以拥有 Server 端对象(必须实现 BnInterface 接口)和 Client 端引用(必须实现 BpInterface 接口)的概念。

1. 服务器端实现

接口 MemoryHeapBase 在 Server 端的实现过程中，可以将所有涉及到的类分为如下所示的三种类型。

- 业务相关类：这是跟匿名共享内存操作相关的类，它们包括 MemoryHeapBase、BnMemoryHeap、IMemoryHeap。
- Binder 进程通信类：即与 Binder 进程通信机制相关的类，包括 IInterface、BnInterface、IBinder、BBinder、ProcessState 和 IPCThreadState。
- 智能指针类：RefBase。

在上述三种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解。在接口 IMemoryBase 中定义了操作匿名共享内存的几个方法，此接口在文件 frameworks\native\include\binder\IMemory.h 中定义，定义代码如下所示：

```
class IMemoryHeap : public IInterface
{
public:
    DECLARE_META_INTERFACE(MemoryHeap);

    // flags returned by getFlags()
    enum {
        READ_ONLY   = 0x00000001
    };

    virtual int         getHeapID() const = 0;
    virtual void*       getBase() const = 0;
    virtual size_t      getSize() const = 0;
    virtual uint32_t    getFlags() const = 0;
    virtual uint32_t    getOffset() const = 0;

    // these are there just for backward source compatibility
    int32_t heapID() const { return getHeapID(); }
```

```
    void*   base() const  { return getBase(); }
    size t  virtualSize() const { return getSize(); }
};
```

在上述定义代码中，有如下 3 个重要的成员函数。

- getHeapID：功能是获得匿名共享内存块的打开文件描述符。
- getBase：功能是获得匿名共享内存块的基地址，通过这个地址，可以在程序里面直接访问这块共享内存。
- getSize：功能是获得匿名共享内存块的大小。

类 BnMemoryHeap 是一个本地对象类，当 Client 端引用请求 Server 端对象执行命令时，Binder 系统就会调用类 BnMemoryHeap 的成员函数 onTransact 执行具体的命令。函数 onTransact 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
status_t BnMemory::onTransact(
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags)
{
    switch(code) {
        case GET_MEMORY: {
            CHECK_INTERFACE(IMemory, data, reply);
            ssize_t offset;
            size_t size;
            reply->writeStrongBinder(getMemory(&offset, &size)->asBinder());
            reply->writeInt32(offset);
            reply->writeInt32(size);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

类 MemoryHeapBase 继承了类 BnMemoryHeap，作为 Binder 机制中的 Server 角色，需要实现 IMemoryBase 接口，主要功能是实现类 IMemoryBase 中列出的成员函数，描述了一块匿名共享内存服务。类在文件 frameworks\native\include\binder\MemoryHeapBase.h 中定义，具体实现代码如下所示：

```
class MemoryHeapBase : public virtual BnMemoryHeap
{
public:
    enum {
        READ_ONLY = IMemoryHeap::READ_ONLY,
        // memory won't be mapped locally, but will be mapped in the remote
        // process.
        DONT_MAP_LOCALLY = 0x00000100,
        NO_CACHING = 0x00000200
    };

    /*
```

```
     * maps the memory referenced by fd. but DOESN'T take ownership
     * of the filedescriptor (it makes a copy with dup()
     */
    MemoryHeapBase(int fd, size_t size, uint32_t flags = 0, uint32_t offset = 0);

    /*
     * maps memory from the given device
     */
    MemoryHeapBase(const char *device, size_t size = 0, uint32_t flags = 0);

    /*
     * maps memory from ashmem, with the given name for debugging
     */
    MemoryHeapBase(size_t size, uint32_t flags = 0, char const *name = NULL);

    virtual ~MemoryHeapBase();

    /* implement IMemoryHeap interface */
    virtual int         getHeapID() const;

    /* virtual address of the heap. returns MAP FAILED in case of error */
    virtual void*       getBase() const;

    virtual size_t      getSize() const;
    virtual uint32_t    getFlags() const;
    virtual uint32_t    getOffset() const;

    const char*         getDevice() const;

    /* this closes this heap -- use carefully */
    void dispose();

    /* this is only needed as a workaround, use only if you know
     * what you are doing */
    status_t setDevice(const char *device) {
        if (mDevice == 0)
            mDevice = device;
        return mDevice ? NO_ERROR : ALREADY_EXISTS;
    }

protected:
    MemoryHeapBase();
    // init() takes ownership of fd
    status_t init(int fd, void *base, int size,
            int flags = 0, const char *device = NULL);

private:
    status_t mapfd(int fd, size_t size, uint32_t offset = 0);
```

```
    int         mFD;    //是一个文件描述符，是在打开设备文件/dev/ashmem后得到的，
                        //能够描述一个匿名共享内存块
    size_t      mSize;  //内存块的大小
    void        *mBase; //内存块的映射地址
    uint32_t    mFlags; //内存块的访问保护位
    const char *mDevice;
    bool        mNeedUnmap;
    uint32_t    mOffset;
};
```

类 MemoryHeapBase 在文件 frameworks\native\libs\binder\MemoryHeapBase.cpp 中实现，其核心功能是包含了一块匿名共享内存。对应的代码如下所示：

```
MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags, char const *name)
    : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
      mDevice(0), mNeedUnmap(false), mOffset(0)
{
    const size_t pagesize = getpagesize();
    size = ((size + pagesize-1) & ~(pagesize-1));
    int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
    ALOGE_IF(fd<0, "error creating ashmem region: %s", strerror(errno));
    if (fd >= 0) {
        if (mapfd(fd, size) == NO_ERROR) {
            if (flags & READ_ONLY) {
                ashmem_set_prot_region(fd, PROT_READ);
            }
        }
    }
}
```

各个参数的具体说明如下所示。

- size：表示要创建的匿名共享内存的大小。
- flags：设置这块匿名共享内存的属性，例如可读写、只读等。
- name：此参数只是作为调试信息使用的，用于标识匿名共享内存的名字，可以是空值。

接下来看 MemoryHeapBase 的成员函数 mapfd，其功能是将得到的匿名共享内存的文件描述符映射到进程地址空间。

函数 mapfd 在文件 frameworks\native\libs\binder\MemoryHeapBase.cpp 中定义，具体实现代码如下所示：

```
status_t MemoryHeapBase::mapfd(int fd, size_t size, uint32_t offset)
{
    if (size == 0) {
        // try to figure out the size automatically
#ifdef HAVE_ANDROID_OS
        // first try the PMEM ioctl
        pmem_region reg;
        int err = ioctl(fd, PMEM_GET_TOTAL_SIZE, &reg);
        if (err == 0)
```

```
            size = reg.len;
#endif
        if (size == 0) { // try fstat
            struct stat sb;
            if (fstat(fd, &sb) == 0)
                size = sb.st_size;
        }
        // if it didn't work, let mmap() fail.
    }

    //条件为 true 时执行系统调用 mmap 来执行内存映射的操作
    if ((mFlags & DONT_MAP_LOCALLY) == 0) {
        void *base = (uint8_t*)mmap(
            0, // 表示由内核来决定这个匿名共享内存文件在进程地址空间的起始位置
            size, // 表示要映射的匿名共享内文件的大小
            PROT_READ|PROT_WRITE, // 表示这个匿名共享内存是可读写的
            MAP_SHARED,
            fd, //指定要映射的匿名共享内存的文件描述符
            offset //表示要从这个文件的哪个偏移位置开始映射
            );
        if (base == MAP_FAILED) {
            ALOGE("mmap(fd=%d, size=%u) failed (%s)",
                    fd, uint32_t(size), strerror(errno));
            close(fd);
            return -errno;
        }
        //ALOGD("mmap(fd=%d, base=%p, size=%lu)", fd, base, size);
        mBase = base;
        mNeedUnmap = true;
    } else  {
        mBase = 0; // not MAP_FAILED
        mNeedUnmap = false;
    }
    mFD = fd;
    mSize = size;
    mOffset = offset;
    return NO_ERROR;
}
```

这样，在调用了函数 mapfd 后，会进入到内核空间的 ashmem 驱动程序模块中，执行函数 ashmem_mmap。

有关函数 ashmem_mmap 的具体实现过程，在 4.2 节的内容中进行过详细讲解。

最后看成员函数 getHeapID、getBase 和 getSize 的具体实现，具体实现代码如下所示：

```
int MemoryHeapBase::getHeapID() const {
    return mFD;
}
void* MemoryHeapBase::getBase() const {
    return mBase;
```

```
}
size_t MemoryHeapBase::getSize() const {
    return mSize;
}
```

2. 客户端实现

接口 MemoryHeapBase 在客户端的实现过程中，可以将所有涉及到的类分为如下所示的 3 种类型。

- 业务相关类：即跟匿名共享内存操作相关的类，包括 BpMemoryHeap、IMemoryHeap。
- Binder 进程通信类：即与 Binder 进程通信机制相关的类，包括 IInterface、BpInterface、IBinder、BpBinder、ProcessState、BpRefBase 和 IPCThreadState。
- 智能指针类：RefBase。

在上述 3 种类型中，Binder 进程通信类和智能指针类将在本书后面的章节中进行讲解，在本章将重点介绍业务相关类。

类 BpMemoryHeap 是类 MemoryHeapBase 在 Client 端进程的远程接口类，Client 端进程从 Service Manager 获得一个 MemoryHeapBase 对象的引用后，会在本地创建一个 BpMemoryHeap 对象来表示这个引用。类 BpMemoryHeap 是从 RefBase 类继承下来的，也要实现 IMemoryHeap 接口，可以与智能指针结合使用。

类 BpMemoryHeap 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
public:
    BpMemoryHeap(const sp<IBinder> &impl);
    virtual ~BpMemoryHeap();

    virtual int getHeapID() const;
    virtual void* getBase() const;
    virtual size_t getSize() const;
    virtual uint32_t getFlags() const;
    virtual uint32_t getOffset() const;

private:
    friend class IMemory;
    friend class HeapCache;

    // for debugging in this module
    static inline sp<IMemoryHeap> find_heap(const sp<IBinder> &binder) {
        return gHeapCache->find_heap(binder);
    }
    static inline void free_heap(const sp<IBinder> &binder) {
        gHeapCache->free_heap(binder);
    }
    static inline sp<IMemoryHeap> get_heap(const sp<IBinder> &binder) {
```

```
        return gHeapCache->get_heap(binder);
    }
    static inline void dump_heaps() {
        gHeapCache->dump_heaps();
    }

    void assertMapped() const;
    void assertReallyMapped() const;

    mutable volatile int32_t mHeapId;
    mutable void        *mBase;
    mutable size_t      mSize;
    mutable uint32_t    mFlags;
    mutable uint32_t    mOffset;
    mutable bool        mRealHeap;
    mutable Mutex       mLock;
};
```

类 BpMemoryHeap 对应的构造函数是 BpMemoryHeap，具体实现代码如下所示：

```
BpMemoryHeap::BpMemoryHeap(const sp<IBinder> &impl)
  : BpInterface<IMemoryHeap>(impl),
  mHeapId(-1), mBase(MAP_FAILED), mSize(0), mFlags(0), mRealHeap(false)
{
}
```

成员函数 getHeapID、getBase 和 getSize 的实现代码如下所示：

```
int BpMemoryHeap::getHeapID() const {
    assertMapped();
    return mHeapId;
}

void* BpMemoryHeap::getBase() const {
    assertMapped();
    return mBase;
}

size_t BpMemoryHeap::getSize() const {
    assertMapped();
    return mSize;
}
```

在使用上述成员函数之前，通过调用函数 assertMapped 来确保在 Client 端已经准备好了匿名共享内存。函数 assertMapped 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
void BpMemoryHeap::assertMapped() const
{
    if (mHeapId == -1) {
        sp<IBinder> binder(const_cast<BpMemoryHeap*>(this)->asBinder());
```

```
        sp<BpMemoryHeap> heap(static_cast<BpMemoryHeap*>(find_heap(binder).get()));
        heap->assertReallyMapped();
        if (heap->mBase != MAP_FAILED) {
            Mutex::Autolock _l(mLock);
            if (mHeapId == -1) {
                mBase   = heap->mBase;
                mSize   = heap->mSize;
                android_atomic_write(dup(heap->mHeapId), &mHeapId);
            }
        } else {
            // something went wrong
            free_heap(binder);
        }
    }
}
```

类 HeapCache 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
class HeapCache : public IBinder::DeathRecipient
{
public:
    HeapCache();
    virtual ~HeapCache();

    virtual void binderDied(const wp<IBinder> &who);

    sp<IMemoryHeap> find_heap(const sp<IBinder> &binder);
    void free_heap(const sp<IBinder> &binder);
    sp<IMemoryHeap> get_heap(const sp<IBinder> &binder);
    void dump_heaps();

private:
    // For IMemory.cpp
    struct heap_info_t {
        sp<IMemoryHeap> heap;
        int32_t         count;
    };

    void free_heap(const wp<IBinder> &binder);

    Mutex mHeapCacheLock;
    KeyedVector<wp<IBinder>, heap_info_t> mHeapCache;
};
```

在上述代码中，定义了成员变量 mHeapCache，功能是维护进程内的所有 BpMemoryHeap 对象。另外还提供了函数 find_heap 和函数 get_heap 来查找内部所维护的 BpMemoryHeap 对象，这两个函数的具体说明如下所示。

- 函数 find_heap：如果在 mHeapCache 找不到相应的 BpMemoryHeap 对象，则把

BpMemoryHeap 对象加入到 mHeapCache 中。

- 函数 get_heap：不会自动把 BpMemoryHeap 对象加入到 mHeapCache 中。

接下来看函数 find_heap，首先以传进来的参数 binder 作为关键字在 mHeapCache 中查找，查找是否存在对应的 heap_info 对象 info。

- 如果有：增加引用计数 info.count 的值，表示此 BpBinder 对象多了一个使用者。
- 如果没有：创建一个，放到 mHeapCache 中的 heap_info 对象 info 中。

函数 find_heap 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
sp<IMemoryHeap> HeapCache::find_heap(const sp<IBinder> &binder)
{
   Mutex::Autolock _l(mHeapCacheLock);
   ssize_t i = mHeapCache.indexOfKey(binder);
   if (i>=0) {
      heap_info_t& info = mHeapCache.editValueAt(i);
      ALOGD_IF(VERBOSE,
            "found binder=%p, heap=%p, size=%d, fd=%d, count=%d",
            binder.get(), info.heap.get(),
            static_cast<BpMemoryHeap*>(info.heap.get())->mSize,
            static_cast<BpMemoryHeap*>(info.heap.get())->mHeapId,
            info.count);
      android_atomic_inc(&info.count);
      return info.heap;
   } else {
      heap info t info;
      info.heap = interface_cast<IMemoryHeap>(binder);
      info.count = 1;
      //ALOGD("adding binder=%p, heap=%p, count=%d",
      //     binder.get(), info.heap.get(), info.count);
      mHeapCache.add(binder, info);
      return info.heap;
   }
}
```

由上述实现代码可知，函数 find_heap 是 BpMemoryHeap 的成员函数，能够调用全局变量 gHeapCache 执行查找的操作。对应的实现代码如下所示：

```
class BpMemoryHeap : public BpInterface<IMemoryHeap>
{
   ...
private:
   static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
      return gHeapCache->find_heap(binder);
   }
   ...
```

通过调用函数 find_heap，得到 BpMemoryHeap 对象中的函数 assertReallyMapped，这样可以确认它内部的匿名共享内存是否已经映射到进程空间。函数 assertReallyMapped 在文件

frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
void BpMemoryHeap::assertReallyMapped() const
{
    if (mHeapId == -1) {

        // remote call without mLock held, worse case scenario, we end up
        // calling transact() from multiple threads, but that's not a problem,
        // only mmap below must be in the critical section.

        Parcel data, reply;
        data.writeInterfaceToken(IMemoryHeap::getInterfaceDescriptor());
        status_t err = remote()->transact(HEAP_ID, data, &reply);
        int parcel_fd = reply.readFileDescriptor();
        ssize_t size = reply.readInt32();
        uint32_t flags = reply.readInt32();
        uint32_t offset = reply.readInt32();

        ALOGE_IF(err, "binder=%p transaction failed fd=%d, size=%ld, err=%d (%s)",
                asBinder().get(), parcel_fd, size, err, strerror(-err));

        int fd = dup(parcel_fd);
        ALOGE_IF(fd==-1, "cannot dup fd=%d, size=%ld, err=%d (%s)",
                parcel_fd, size, err, strerror(errno));

        int access = PROT_READ;
        if (!(flags & READ_ONLY)) {
            access |= PROT_WRITE;
        }

        Mutex::Autolock _l(mLock);
        if (mHeapId == -1) {
            mRealHeap = true;
            mBase = mmap(0, size, access, MAP_SHARED, fd, offset);
            if (mBase == MAP_FAILED) {
                ALOGE("cannot map BpMemoryHeap (binder=%p), size=%ld, fd=%d (%s)",
                        asBinder().get(), size, fd, strerror(errno));
                close(fd);
            } else {
                mSize = size;
                mFlags = flags;
                mOffset = offset;
                android_atomic_write(fd, &mHeapId);
            }
        }
    }
}
```

4.3.2　接口MemoryBase

接口 MemoryBase 是建立在接口 MemoryHeapBase 的基础上的，两者都可以作为一个 Binder 对象，在进程之间实现数据共享。

1. 在Server端的实现

首先分析 MemoryBase 类在 Server 端的实现，MemoryBase 在 Server 端只是简单地封装了 MemoryHeapBase 的实现。类 MemoryBase 在 Server 端的实现跟类 MemoryHeapBase 在 Server 端的实现类似，只需在整个类图结构中实现如下转换即可：

- 把类 IMemory 换成类 IMemoryHeap。
- 把类 BnMemory 换成类 BnMemoryHeap。
- 把类 MemoryBase 换成类 MemoryHeapBase。

类 IMemory 在文件 frameworks\native\include\binder\IMemory.h 中实现，功能是定义类 MemoryBase 所需要的实现接口。类 IMemory 的实现代码如下所示：

```
class IMemory : public IInterface
{
public:
    DECLARE_META_INTERFACE(Memory);
    virtual sp<IMemoryHeap> getMemory(ssize_t *offset=0, size_t *size=0) const = 0;
    // helpers
    void* fastPointer(const sp<IBinder> &heap, ssize_t offset) const;
    void* pointer() const;
    size_t size() const;
    ssize_t offset() const;
};
```

在类 IMemory 中定义了如下所示的成员函数。

- getMemory：获取内部的 MemoryHeapBase 对象的 IMemoryHeap 接口。
- pointer()：获取内部所维护的匿名共享内存的基地址。
- size()：获取内部所维护的匿名共享内存的大小。
- offset()：获取内部所维护的匿名共享内存在整个匿名共享内存中的偏移量。

类 IMemory 在本身定义过程中实现了三个成员函数：pointer、size 和 offset，其子类 MemoryBase 只需实现成员函数 getMemory 即可。类 IMemory 的具体实现在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
void* IMemory::pointer() const {
    ssize_t offset;
    sp<IMemoryHeap> heap = getMemory(&offset);
    void *const base = heap!=0? heap->base() : MAP_FAILED;
    if (base == MAP_FAILED)
        return 0;
    return static_cast<char*>(base) + offset;
}
```

```
size t IMemory::size() const {
    size_t size;
    getMemory(NULL, &size);
    return size;
}
ssize_t IMemory::offset() const {
    ssize_t offset;
    getMemory(&offset);
    return offset;
}
```

类 MemoryBase 是一个本地 Binder 对象类，在文件 frameworks\native\include\binder\MemoryBase.h 中声明，具体定义代码如下所示：

```
class MemoryBase : public BnMemory
{
public:
    MemoryBase(const sp<IMemoryHeap> &heap, ssize_t offset, size_t size);
    virtual ~MemoryBase();
    virtual sp<IMemoryHeap> getMemory(ssize_t *offset, size_t *size) const;
protected:
    size_t getSize() const { return mSize; }
    ssize t getOffset() const { return mOffset; }
    const sp<IMemoryHeap>& getHeap() const { return mHeap; }
private:
    size_t          mSize;
    ssize_t         mOffset;
    sp<IMemoryHeap> mHeap;
};
}; // namespace android
#endif // ANDROID_MEMORY_BASE_H
```

类 MemoryBase 的具体实现在文件 frameworks\native\libs\binder\MemoryBase.cpp 中定义，具体实现代码如下所示：

```
MemoryBase::MemoryBase(const
 sp<IMemoryHeap> &heap, //指向 MemoryHeapBase 对象，真正的匿名共享内存就是由它来维护的
 ssize_t offset, //表示这个 MemoryBase 对象所要维护的这部分匿名共享内存
                 //在整个匿名共享内存块中的起始位置
 size_t size //表示这个 MemoryBase 对象所要维护的这部分匿名共享内存的大小
) : mSize(size), mOffset(offset), mHeap(heap)
{
}
//功能是返回内部的 MemoryHeapBase 对象的 IMemoryHeap 接口
//如果传进来的参数 offset 和 size 不为 NULL
//会把其内部维护的这部分匿名共享内存在整个匿名共享内存块中的偏移位置
//以及这部分匿名共享内存的大小返回给调用者
sp<IMemoryHeap> MemoryBase::getMemory(ssize_t *offset, size_t *size) const
{
```

```
    if (offset) *offset = mOffset;
    if (size)  *size = mSize;
    return mHeap;
}
```

2. MemoryBase类在Client端的实现

再来看 MemoryBase 类在 Client 端的实现，类 MemoryBase 在 Client 端的实现与类 MemoryHeapBase 在 Client 端的实现类似，只需要进行如下所示的类转换，即可成为 MemoryHeapBase 在 Client 端的实现：

- 把类 IMemory 换成类 IMemoryHeap。
- 把类 BpMemory 换成类 BpMemoryHeap。

类 BpMemory 用于描述类 MemoryBase 服务的代理对象，在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
class BpMemory : public BpInterface<IMemory>
{
public:
    BpMemory(const sp<IBinder> &impl);
    virtual ~BpMemory();
    virtual sp<IMemoryHeap> getMemory(ssize_t *offset=0, size_t *size=0) const;

private:
    mutable sp<IMemoryHeap> mHeap; //类型为 IMemoryHeap，它指向的是一个 BpMemoryHeap 对象
    mutable ssize_t mOffset;//表示 BpMemory 对象所要维护的匿名共享内存
                            //在整个匿名共享内存块中的起始位置
    mutable size_t mSize; //表示这个 BpMemory 对象所要维护的这部分匿名共享内存的大小
};
```

类 BpMemory 中的成员函数 getMemory 在文件 frameworks\native\libs\binder\IMemory.cpp 中定义，具体实现代码如下所示：

```
sp<IMemoryHeap> BpMemory::getMemory(ssize_t *offset, size_t *size) const
{
    if (mHeap == 0) {
        Parcel data, reply;
        data.writeInterfaceToken(IMemory::getInterfaceDescriptor());
        if (remote()->transact(GET_MEMORY, data, &reply) == NO_ERROR) {
            sp<IBinder> heap = reply.readStrongBinder();
            ssize_t o = reply.readInt32();
            size_t s = reply.readInt32();
            if (heap != 0) {
                mHeap = interface_cast<IMemoryHeap>(heap);
                if (mHeap != 0) {
                    mOffset = o;
                    mSize = s;
                }
```

```
            }
        }
    }
    if (offset) *offset = mOffset;
    if (size) *size = mSize;
    return mHeap;
}
```

如果成员变量 mHeap 的值为 NULL，表示此 BpMemory 对象还没有建立好匿名共享内存，此时会调用一个 Binder 进程去 Server 端请求匿名共享内存信息。通过引用信息中的 Server 端的 MemoryHeapBase 对象的引用 heap，可以在 Client 端进程中创建一个 BpMemoryHeap 远程接口，最后将这个 BpMemoryHeap 远程接口保存在成员变量 mHeap 中，同时从 Server 端获得的信息还包括这块匿名共享内存在整个匿名共享内存中的偏移位置以及大小。

4.4　分析Java访问接口层

分析完匿名共享内存的 C++访问接口层后，从本节开始，分析其 Java 访问接口层的实现过程。在 Android 应用程序框架层中，通过使用接口 MemoryFile 来封装匿名共享内存文件的创建和使用。接口 MemoryFile 在文件 frameworks/base/core/java/android/os/MemoryFile.java 中定义，具体代码如下所示：

```
public class MemoryFile
{
    private static String TAG = "MemoryFile";

    // mmap(2) protection flags from <sys/mman.h>
    private static final int PROT_READ = 0x1;
    private static final int PROT_WRITE = 0x2;

    private static native FileDescriptor native_open(String name, int length)
      throws IOException;
    // returns memory address for ashmem region
    private static native int native_mmap(FileDescriptor fd, int length, int mode)
      throws IOException;
    private static native void native_munmap(int addr, int length) throws IOException;
    private static native void native_close(FileDescriptor fd);
    private static native int native_read(FileDescriptor fd, int address, byte[] buffer,
      int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
    private static native void native_write(FileDescriptor fd, int address, byte[] buffer,
      int srcOffset, int destOffset, int count, boolean isUnpinned)
      throws IOException;
    private static native void native_pin(FileDescriptor fd, boolean pin)
      throws IOException;
    private static native int native_get_size(FileDescriptor fd)
      throws IOException;
```

```
    private FileDescriptor mFD;       // ashmem file descriptor
    private int mAddress;   // address of ashmem memory
    private int mLength;   // total length of our ashmem region
    private boolean mAllowPurging = false;  // true if our ashmem region is unpinned

    /**
     * Allocates a new ashmem region. The region is initially not purgable.
     *
     * @param name optional name for the file (can be null).
     * @param length of the memory file in bytes.
     * @throws IOException if the memory file could not be created.
     */
    public MemoryFile(String name, int length) throws IOException {
        mLength = length;
        mFD = native_open(name, length);
        if (length > 0) {
            mAddress = native_mmap(mFD, length, PROT_READ | PROT_WRITE);
        } else {
            mAddress = 0;
        }
    }
}
```

在上述代码中，构造方法 MemoryFile 以指定的字符串调用了 JNI 方法 native_open，目的是建立一个匿名共享内存文件，这样可以得到一个文件描述符。然后使用这个文件描述符为参数调用 JNI 方法 natvie_mmap，并把匿名共享内存文件映射到进程空间中，这样就可以通过映射得到地址空间的方式直接访问内存数据。

再看 JNI 函数 android_os_MemoryFile_get_size，此函数在文件 frameworks\base\core\jni\android_os_MemoryFile.cpp 中定义，具体实现代码如下所示：

```
static jint android_os_MemoryFile_get_size(JNIEnv *env, jobject clazz,
  jobject fileDescriptor) {
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    // Use ASHMEM_GET_SIZE to find out if the fd refers to an ashmem region.
    // ASHMEM_GET_SIZE should succeed for all ashmem regions, and the kernel
    // should return ENOTTY for all other valid file descriptors
    int result = ashmem_get_size_region(fd);
    if (result < 0) {
        if (errno == ENOTTY) {
            // ENOTTY means that the ioctl does not apply to this object,
            // i.e., it is not an ashmem region.
            return (jint) -1;
        }
        // Some other error, throw exception
        jniThrowIOException(env, errno);
        return (jint) -1;
    }
```

```
    return (jint) result;
}
```

再看 JNI 函数 android_os_MemoryFile_open，此函数在文件 frameworks\base\core\jni\android_os_MemoryFile.cpp 中定义，具体实现代码如下所示：

```
static jobject android_os_MemoryFile_open(JNIEnv *env, jobject clazz,
  jstring name, jint length)
{
    const char *namestr = (name? env->GetStringUTFChars(name, NULL) : NULL);

    int result = ashmem_create_region(namestr, length);

    if (name)
        env->ReleaseStringUTFChars(name, namestr);

    if (result < 0) {
        jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
        return NULL;
    }

    return jniCreateFileDescriptor(env, result);
}
```

再看 JNI 函数 android_os_MemoryFile_mmap，此函数在文件 frameworks\base\core\jni\android_os_MemoryFile.cpp 中定义，具体实现代码如下所示：

```
static jint android_os_MemoryFile_mmap(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，再看类 MemoryFile 的成员函数 readBytes，功能是读取某一块匿名共享内存的内容。具体实现代码如下所示：

```
public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
  throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't read from deactivated memory file.");
    }
    if (destOffset<0 || destOffset>buffer.length || count<0
      || count>buffer.length-destOffset
      || srcOffset<0 || srcOffset>mLength
      || count>mLength-srcOffset) {
        throw new IndexOutOfBoundsException();
```

```
    }
    return native read(
      mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，再看类 MemoryFile 的成员函数 writeBytes，功能是写入某一块匿名共享内存的内容。具体实现代码如下所示：

```
public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
  throws IOException {
    if (isDeactivated()) {
        throw new IOException("Can't write to deactivated memory file.");
    }
    if (srcOffset<0 || srcOffset>buffer.length || count<0
      || count>buffer.length-srcOffset
      || destOffset<0 || destOffset>mLength
      || count>mLength-destOffset) {
        throw new IndexOutOfBoundsException();
    }
    native_write(
      mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

在文件 frameworks/base/core/java/android/os/MemoryFile.java 中，再看类 MemoryFile 的成员函数 isDeactivated，功能是保证匿名共享内存已经被映射到进程的地址空间中。具体实现代码如下所示：

```
void deactivate() {
    if (!isDeactivated()) {
        try {
            native_munmap(mAddress, mLength);
            mAddress = 0;
        } catch (IOException ex) {
            Log.e(TAG, ex.toString());
        }
    }
}
private boolean isDeactivated() {
    return mAddress == 0;
}
```

JNI 函数 native_read 和 native_write 分别由位于 C++层的函数 android_os_MemoryFile_read 和 android_os_MemoryFile_write 实现，这两个 C++的函数在文件 frameworks\base\core\jni\android_os_MemoryFile.cpp 中定义，具体实现代码如下所示：

```
static jint android_os_MemoryFile_read(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset,
  jint destOffset, jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
```

```
    if (unpinned && ashmem_pin_region(fd, 0, 0)==ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->SetByteArrayRegion(
      buffer, destOffset, count, (const jbyte*)address + srcOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
static jint android_os_MemoryFile_write(JNIEnv *env, jobject clazz,
  jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset,
  jint destOffset, jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0)==ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }
    env->GetByteArrayRegion(
      buffer, srcOffset, count, (jbyte*)address + destOffset);
    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

4.5 内存优化机制

在 Android 系统中，使用垃圾回收机制的方式达到节约内存的目的，并最终实现提高手机的处理效率的目的。在本节的内容中，将详细讲解 Android 系统中的垃圾回收机制的知识，为读者步入本书后面知识的学习打下基础。

4.5.1 sp和wp简析

在传统的 C++编程语言中，指针一直是程序员的最大学习障碍。指针比较复杂，一旦使用不当，就会造成内存泄漏的问题。例如用 new 新建一个对象并使用完之后，经常忘记 delete(删除)这个对象，长期下去会造成系统崩溃。在 Android 系统中，因为其运行时库这一层代码是用 C++来编写的，所以也会因为使用指针的原因而造成内存泄漏问题。为此 Android 特意提供了智能指针机制，通过使用 sp 命令和 wp 命令来解决指针问题。其实 sp 和 wp 就是 Android 为其 C++实现的自动垃圾回收机制。如果具体到内部实现，sp 和 wp 实际上只是一个实现垃圾回收

功能的接口而已，而真正实现垃圾回收的是 refbase 这个基类。这部分代码位于如下文件中：

```
/frameworks/base/include/utils/RefBase.h
```

在此，所有的类都会虚继承于 refbase 类，因为它实现了 Android 垃圾回收所需要的所有 function，因此实际上所有的对象声明出来以后都具备了自动释放自己的能力，也就是说，实际上智能指针就是我们的对象本身，它会维持一个对本身强引用和弱引用的计数，一旦强引用计数为 0，它就会释放掉自己。

(1)　sp

sp 实际上不是 smart pointer 的缩写，而是 strong pointer，它实际上内部只包含了一个指向对象的指针而已。我们可以简单地看看 sp 的一个构造函数：

```
template<typename T>
sp<T>::sp(T *other) : m_ptr(other)
{
    if (other) other->incStrong(this);
}
```

比如说我们声明一个对象：

```
sp<CameraHardwareInterface> hardware(new CameraHal());
```

实际上 sp 指针对本身没有进行什么操作，就是一个指针的赋值，包含了一个指向对象的指针，但是对象会对对象本身增加一个强引用计数，这个 incStrong 的实现就在 refbase 类里面。新 new 出来一个 CameraHal 对象，将它的值给 sp<CameraHardwareInterface>的时候，它的强引用计数就会从 0 变为 1。因此每次将对象赋值给一个 sp 指针的时候，对象的强引用计数都会加 1，下面我们再看看 sp 的析构函数：

```
template<typename T>
sp<T>::~sp()
{
    if (m_ptr) m_ptr->decStrong(this);
}
```

实际上，每次删除一个 sp 对象的时候，sp 指针指向的对象的强引用计数就会减 1，当对象的强引用计数为 0 的时候，这个对象就会被自动释放掉。

(2)　wp

我们再看 wp，wp 就是 weak pointer 的缩写，弱引用指针的原理，就是为了应用 Android 垃圾回收来减少对那些胖对象对内存的占用，我们首先来看 wp 的一个构造函数：

```
wp<T>::wp(T *other) : m_ptr(other)
{
    if (other) m_refs = other->createWeak(this);
}
```

它和 sp 一样，实际上也就是仅仅对指针进行了赋值而已，对象本身会增加一个对自身的弱引用计数，同时 wp 还包含一个 m_ref 指针，这个指针主要是用来将 wp 升级为 sp：

```
template<typename T>
sp<T> wp<T>::promote() const
{
    return sp<T>(m_ptr, m_refs);
}
template< typename T>
sp<T>::sp(T *p, weakref_type *refs)
  : m_ptr((p && refs->attemptIncStrong(this))? p : 0)
{
}
```

实际上，我们对 wp 指针唯一能做的就是将 wp 指针升级为一个 sp 指针，然后判断是否升级成功，如果成功，说明对象依旧存在，如果失败，说明对象已经被释放掉了。wp 指针在单例中使用很多，确保 mhardware 对象只有一个，例如：

```
wp<CameraHardwareInterface> CameraHardwareStub::singleton;
sp<CameraHardwareInterface> CameraHal::createInstance()
{
    LOG_FUNCTION_NAME
    if (singleton != 0) {
        sp<CameraHardwareInterface> hardware = singleton.promote();
        if (hardware != 0) {
            return hardware;
        }
    }
    sp<CameraHardwareInterface> hardware(new CameraHal()); //强引用加 1
    singleton = hardware; //弱引用加 1
    return hardware; //赋值构造函数，强引用加 1
}
//hardware 被删除，强引用减 1
```

4.5.2 详解智能指针

在 Android 的源代码中，经常会看到形如：sp<xxx>、wp<xxx>形式的类型定义，这其实是 Android 中的智能指针。Android 的智能指针相关的源代码在如下两个文件中：

```
frameworks/base/include/utils/RefBase.h
frameworks/base/libs/utils/RefBase.cpp
```

涉及的类以及类之间的关系如图 4-3 所示。

Android 中定义了三种智能指针类型，分别是强指针 sp(Strong Pointer)、弱指针(Weak Pointer)和轻量级指针(Light Pointer)。其实称为强引用和弱引用更合适一些。强指针与一般意义的智能指针概念相同，通过引用计数来记录有多少使用者在使用一个对象，如果所有使用者都放弃了对该对象的引用，则该对象将被自动销毁。

弱指针也指向一个对象，但是弱指针仅仅记录该对象的地址，不能通过弱指针来访问该对象，也就是说，不能通过弱指针来调用对象的成员函数或访问对象的成员变量。要想访问弱指针所指向的对象，需首先将弱指针升级为强指针(通过 wp 类所提供的 promote()方法)。弱指针

所指向的对象是有可能在其他地方被销毁的，如果对象已经被销毁，wp 的 promote()方法将返回空指针，这样就能避免出现地址访问错误的情况。

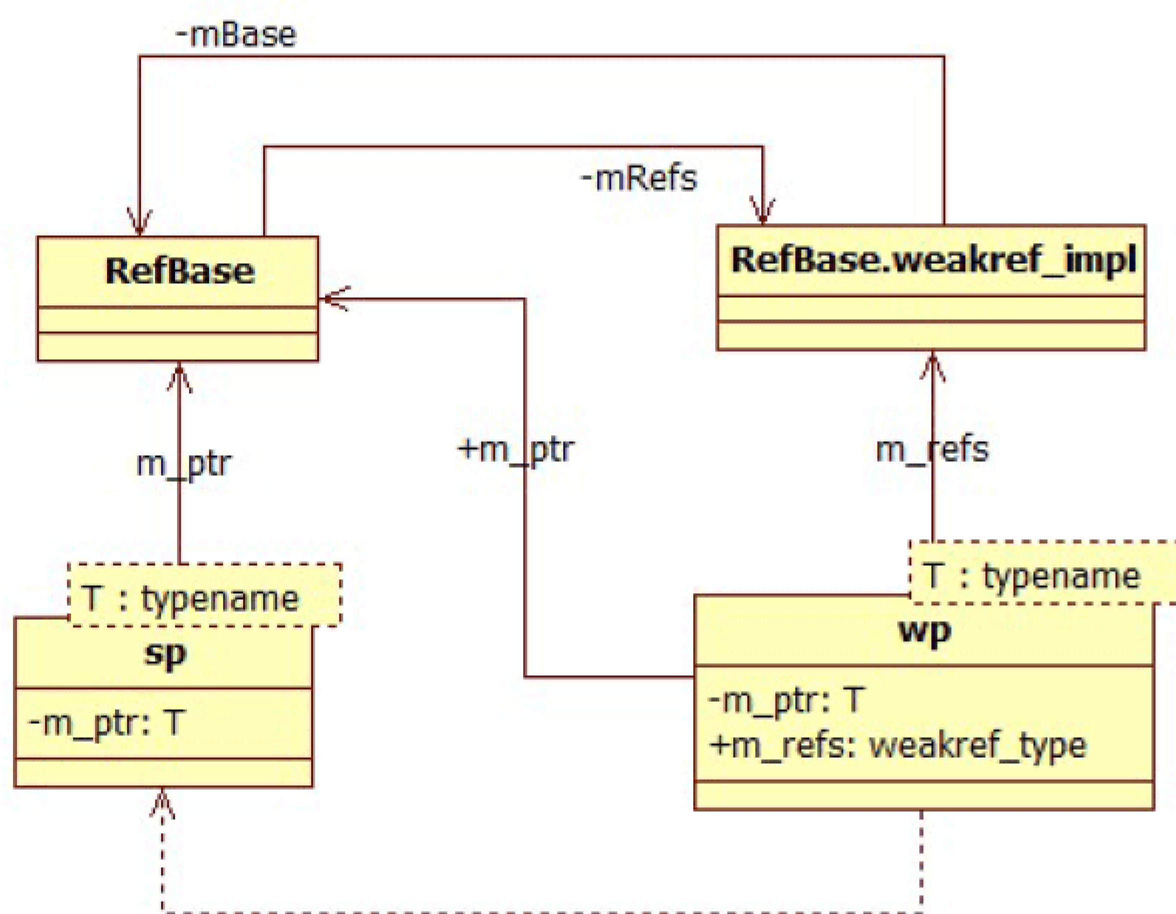


图 4-3　智能指针相关类的关系

究竟指针是怎么做到这一点的呢？其实一点也不复杂，原因就在于每一个可以被智能指针引用的对象，都同时被附加了另外一个 weakref_impl 类型的对象，这个对象中负责记录对象的强指针引用计数和弱指针引用计数。这个对象是智能指针的实现内部使用的，智能指针的使用者看不到这个对象。弱指针操作的就是这个对象，只有当强引用计数和弱引用计数都为 0 时，这个对象才会被销毁。

接下来开始分析到底该怎样使用智能指针。假设现在有一个 MyClass 类，如果要使用智能指针来引用这个类的对象，那么这个类需满足下列两个前提条件。

(1)　这个类是基类 RefBase 的子类或间接子类。

(2)　这个类必须定义虚构造函数，即它的构造函数需要这样定义：

```
virtual ~MyClass();
```

满足了上述条件的类后，就可以定义智能指针，定义方法和普通指针类似。比如普通指针是这样定义的：

```
MyClass *p_obj;
```

智能指针是这样定义：

```
sp<MyClass> p_obj;
```

注意不要定义成 sp<MyClass>* p_obj。初学者容易犯这种错误，这样实际上相当于定义了一个指针的指针。尽管在语法上没有问题，但是最好永远不要使用这样的定义。

定义一个智能指针的变量后，就可以像普通指针那样使用它了，包括赋值、访问对象成员、作为函数的返回值、作为函数的参数等。例如：

```
p_obj = new MyClass(); // 注意不要写成 p_obj = new sp<MyClass>;
sp<MyClass> p_obj2 = p_obj;
p_obj->func();
```

```
p_obj = create_obj();
some_func(p_obj);
```

注意不要试图 delete 一个智能指针，即不要执行 delete p_obj 操作。我们无须担心对象的销毁问题，智能指针的最大作用就是自动销毁不再使用的对象。当不需要再使用一个对象后，只需直接将指针赋值为 NULL 即可：

```
p_obj = NULL;
```

上面说的都是强指针，弱指针的定义方法与强指针类似，但是不能通过弱指针来访问对象的成员。下面是弱指针的示例：

```
wp<MyClass> wp obj = new MyClass();
p_obj = wp_obj.promote(); // 升级为强指针。不过这里要用.而不是->，真是有负其指针之名啊
wp_obj = NULL;
```

由此可见，智能指针用起来是很方便，在一般情况下最好使用智能指针来代替普通指针。但是需要知道，一个智能指针其实是一个对象，而不是一个真正的指针，因此其运行效率是远远比不上普通指针的。所以在对运行效率敏感的地方，最好还是不要使用智能指针为好。

4.5.3 轻量级指针

在 Android 系统中，轻量级指针通过引用计数技术来维护对象的声明周期。支持轻量级指针的对象必须继承自基类 LightRefBase，类 LightRefBase 在文件 frameworks\native\include\utils\RefBase.h 中定义，具体实现代码如下所示：

```
template <class T>
class LightRefBase
{
public:
    inline LightRefBase() : mCount(0) { }
    inline void incStrong(const void *id) const {
        android_atomic_inc(&mCount);
    }
    inline void decStrong(const void *id) const {
        if (android_atomic_dec(&mCount) == 1) {
            delete static_cast<const T*>(this);
        }
    }
    //! DEBUGGING ONLY: Get current strong ref count.
    inline int32_t getStrongCount() const {
        return mCount;
    }
    typedef LightRefBase<T> basetype;
protected:
    inline ~LightRefBase() { }
private:
    friend class ReferenceMover;
    inline static void moveReferences(void *d, void const *s, size_t n,
```

```
         const ReferenceConverterBase &caster) { }
private:
   mutable volatile int32_t mCount;
};
```

由上述代码可以看出，类 LightRefBase 只引用一个计数器成员变量 mCount，其初始化值为 0。另外，类 LightRefBase 还通过成员函数 incStrong 和 decStrong 维护引用计数器的值，这两个函数被智能指针调用。在函数 decStrong 中，如果当前引用计数值为 1，那么减 1 后就会变为 0，这表示 delete(删除)这个对象。

在 Android 系统中，和 LightRefBase 引用计数配套使用的智能指针类是 sp，sp 是轻量级指针的实现类。在文件 frameworks\native\include\utils\RefBase.h 中，sp 的具体实现代码如下所示：

```
template <typename T>
class sp
{
public:
    typedef typename RefBase::weakref_type weakref_type;

    inline sp() : m_ptr(0) { }

    sp(T *other); //T 表示对象的实际类型,
    sp(const sp<T> &other);
    template<typename U> sp(U *other);
    template<typename U> sp(const sp<U> &other);

    ~sp();

    // Assignment

    sp& operator = (T *other);
    sp& operator = (const sp<T> &other);

    template<typename U> sp &operator = (const sp<U> &other);
    template<typename U> sp &operator = (U *other);

    //! Special optimization for use by ProcessState (and nobody else).
    void force_set(T *other);

    // Reset

    void clear();

    // Accessors

    inline  T&     operator* () const  { return *m_ptr; }
    inline  T*     operator-> () const { return m_ptr;  }
    inline  T*     get() const        { return m_ptr; }
```

```
    // Operators

    COMPARE(==)
        COMPARE(!=)
        COMPARE(>)
        COMPARE(<)
        COMPARE(<=)
        COMPARE(>=)

private:
    template<typename Y> friend class sp;
    template<typename Y> friend class wp;

    // Optimization for wp::promote().
    sp(T *p, weakref_type *refs);

    T *m_ptr;
};
```

类 sp 有如下两个构造函数：

- 普通构造函数。
- 拷贝构造函数。

上述两个构造函数在文件 frameworks\native\include\utils\RefBase.h 中实现，具体实现代码如下所示：

```
template<typename T>
sp<T>::sp(T *other)
    : m_ptr(other)
{
    if (other) other->incStrong(this);
}

template<typename T>
sp<T>::sp(const sp<T> &other)
    : m_ptr(other.m_ptr)
{
    if (m_ptr) m_ptr->incStrong(this);
}
```

类 sp 中包含了析构函数，功能是调用 m_ptr 的成员函数 decStrong 减少对象的引用计数值。函数 decStrong 在类 LightRefBase 中定义，当引用计数减 1 后变成 0 时，会自动 delete(删除)这个对象。定义析构函数的实现代码如下所示：

```
template<typename T>
sp<T>::~sp()
{
    if (m_ptr) m_ptr->decStrong(this);
}
```

4.5.4　强指针

在 Android 系统中，强指针使用的引用计数类是 RefBase，类 RefBase 比类 LightRefBase 要复杂。但是其功能与类 LightRefBase 一样，也提供了 incStrong 和 decStrong 成员函数来操作它的引用计数器。类 RefBase 与类 LightRefBase 的最大区别是，它不像类 LightRefBase 一样直接提供一个整型值(mutable volatile int32_t mCount)来维护对象的引用计数。原因是复杂的引用计数技术同时支持强引用计数和弱引用计数。所以在类 RefBase 的具体实现中，强引用计数和弱引用计数功能是通过其成员变量 mRefs 提供的。

类 RefBase 在文件 frameworks\native\include\utils\RefBase.h 中定义，具体实现代码如下：

```
class RefBase
{
public:
    void         incStrong(const void *id) const;
    void         decStrong(const void *id) const;
    void         forceIncStrong(const void *id) const;
    //! DEBUGGING ONLY: Get current strong ref count.
    int32_t       getStrongCount() const;
    class weakref_type
    {
    public:
       RefBase*          refBase() const;

       void              incWeak(const void *id);
       void              decWeak(const void *id);

       // acquires a strong reference if there is already one.
       bool              attemptIncStrong(const void *id);

       // acquires a weak reference if there is already one.
       // This is not always safe. see ProcessState.cpp and BpBinder.cpp
       // for proper use.
       bool              attemptIncWeak(const void *id);
       //! DEBUGGING ONLY: Get current weak ref count.
       int32_t            getWeakCount() const;
       //! DEBUGGING ONLY: Print references held on object.
       void              printRefs() const;
       //! DEBUGGING ONLY: Enable tracking for this object.
       // enable -- enable/disable tracking
       // retain -- when tracking is enable, if true, then we save a stack trace
       //          for each reference and dereference; when retain == false, we
       //          match up references and dereferences and keep only the
       //          outstanding ones.

       void              trackMe(bool enable, bool retain);
    };
```

```
    weakref_type*   createWeak(const void *id) const;

    weakref_type*   getWeakRefs() const;
    //! DEBUGGING ONLY: Print references held on object.
    inline  void          printRefs() const { getWeakRefs()->printRefs(); }
    //! DEBUGGING ONLY: Enable tracking of object.
    inline  void          trackMe(bool enable, bool retain)
    {
        getWeakRefs()->trackMe(enable, retain);
    }
    typedef RefBase basetype;
protected:
                          RefBase();
    virtual               ~RefBase();
    //! Flags for extendObjectLifetime()
    enum {
        OBJECT_LIFETIME_STRONG  = 0x0000,
        OBJECT_LIFETIME_WEAK    = 0x0001,
        OBJECT_LIFETIME_MASK    = 0x0001
    };
    void          extendObjectLifetime(int32_t mode);

    //! Flags for onIncStrongAttempted()
    enum {
        FIRST_INC_STRONG = 0x0001
    };

    virtual void          onFirstRef();
    virtual void          onLastStrongRef(const void *id);
    virtual bool          onIncStrongAttempted(uint32_t flags, const void *id);
    virtual void          onLastWeakRef(const void *id);
private:
    friend class ReferenceMover;
    static void moveReferences(void *d, void const *s, size_t n,
            const ReferenceConverterBase &caster);
private:
    friend class weakref_type;
    class weakref_impl;
    RefBase(const RefBase &o);
    RefBase& operator=(const RefBase &o);
    weakref_impl* const mRefs;
};
```

在类 RefBase 中，其成员变量 mRefs 的类型为 weakref_impl 指针。类 RefBase 的具体实现在文件 frameworks\native\libs\utils\RefBase.cpp 中定义，代码如下所示：

```
class RefBase::weakref_impl : public RefBase::weakref_type
{
public:
```

```
    volatile int32_t    mStrong;
    volatile int32_t    mWeak;
    RefBase* const      mBase;
    volatile int32_t    mFlags;

#if !DEBUG_REFS

    weakref_impl(RefBase *base)
        : mStrong(INITIAL_STRONG_VALUE)
        , mWeak(0)
        , mBase(base)
        , mFlags(0)
    {
    }

    void addStrongRef(const void* /*id*/) { }
    void removeStrongRef(const void* /*id*/) { }
    void renameStrongRefId(const void* /*old_id*/, const void* /*new_id*/) { }
    void addWeakRef(const void* /*id*/) { }
    void removeWeakRef(const void* /*id*/) { }
    void renameWeakRefId(const void* /*old_id*/, const void* /*new_id*/) { }
    void printRefs() const { }
    void trackMe(bool, bool) { }

#else

    weakref_impl(RefBase *base)
        : mStrong(INITIAL_STRONG_VALUE)
        , mWeak(0)
        , mBase(base)
        , mFlags(0)
        , mStrongRefs(NULL)
        , mWeakRefs(NULL)
        , mTrackEnabled(!!DEBUG_REFS_ENABLED_BY_DEFAULT)
        , mRetain(false)
    {
    }

    ~weakref_impl()
    {
        bool dumpStack = false;
        if (!mRetain && mStrongRefs!=NULL) {
            dumpStack = true;
#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("Strong references remain!");
#else
            ALOGE("Strong references remain:");
#endif
            ref_entry *refs = mStrongRefs;
```

```
            while (refs) {
                char inc = refs->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, refs->id, refs->ref);
#if DEBUG_REFS_CALLSTACK_ENABLED
                refs->stack.dump();
#endif
                refs = refs->next;
            }
        }

        if (!mRetain && mWeakRefs!=NULL) {
            dumpStack = true;
#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("Weak references remain:");
#else
            ALOGE("Weak references remain!");
#endif
            ref_entry *refs = mWeakRefs;
            while (refs) {
                char inc = refs->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, refs->id, refs->ref);
#if DEBUG_REFS_CALLSTACK_ENABLED
                refs->stack.dump();
#endif
                refs = refs->next;
            }
        }
        if (dumpStack) {
            ALOGE("above errors at:");
            CallStack stack;
            stack.update();
            stack.dump();
        }
    }

    void addStrongRef(const void *id) {
        //ALOGD_IF(mTrackEnabled,
        //        "addStrongRef: RefBase=%p, id=%p", mBase, id);
        addRef(&mStrongRefs, id, mStrong);
    }

    void removeStrongRef(const void *id) {
        //ALOGD_IF(mTrackEnabled,
        //        "removeStrongRef: RefBase=%p, id=%p", mBase, id);
        if (!mRetain) {
            removeRef(&mStrongRefs, id);
        } else {
            addRef(&mStrongRefs, id, -mStrong);
        }
```

```
    }

    void renameStrongRefId(const void *old_id, const void *new_id) {
        //ALOGD_IF(mTrackEnabled,
        //        "renameStrongRefId: RefBase=%p, oid=%p, nid=%p",
        //        mBase, old_id, new_id);
        renameRefsId(mStrongRefs, old_id, new_id);
    }

    void addWeakRef(const void *id) {
        addRef(&mWeakRefs, id, mWeak);
    }

    void removeWeakRef(const void *id) {
        if (!mRetain) {
            removeRef(&mWeakRefs, id);
        } else {
            addRef(&mWeakRefs, id, -mWeak);
        }
    }

    void renameWeakRefId(const void *old_id, const void *new_id) {
        renameRefsId(mWeakRefs, old_id, new_id);
    }

    void trackMe(bool track, bool retain)
    {
        mTrackEnabled = track;
        mRetain = retain;
    }

    void printRefs() const
    {
        String8 text;

        {
            Mutex::Autolock _l(mMutex);
            char buf[128];
            sprintf(buf, "Strong references on RefBase %p (weakref_type %p):\n", mBase, this);
            text.append(buf);
            printRefsLocked(&text, mStrongRefs);
            sprintf(buf, "Weak references on RefBase %p (weakref_type %p):\n", mBase, this);
            text.append(buf);
            printRefsLocked(&text, mWeakRefs);
        }

        {
            char name[100];
            snprintf(name, 100, "/data/%p.stack", this);
```

```
            int rc = open(name, O_RDWR | O_CREAT | O_APPEND);
            if (rc >= 0) {
                write(rc, text.string(), text.length());
                close(rc);
                ALOGD("STACK TRACE for %p saved in %s", this, name);
            }
            else ALOGE("FAILED TO PRINT STACK TRACE for %p in %s: %s", this,
                      name, strerror(errno));
        }
    }

private:
    struct ref_entry
    {
        ref_entry *next;
        const void *id;
#if DEBUG_REFS_CALLSTACK_ENABLED
        CallStack stack;
#endif
        int32_t ref;
    };

    void addRef(ref_entry **refs, const void *id, int32_t mRef)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);

            ref_entry *ref = new ref_entry;
            // Reference count at the time of the snapshot, but before the
            // update.  Positive value means we increment, negative--we
            // decrement the reference count.
            ref->ref = mRef;
            ref->id = id;
#if DEBUG_REFS_CALLSTACK_ENABLED
            ref->stack.update(2);
#endif
            ref->next = *refs;
            *refs = ref;
        }
    }

    void removeRef(ref_entry **refs, const void *id)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);

            ref_entry *const head = *refs;
            ref_entry *ref = head;
            while (ref != NULL) {
```

```
                if (ref->id == id) {
                    *refs = ref->next;
                    delete ref;
                    return;
                }
                refs = &ref->next;
                ref = *refs;
            }

#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("RefBase: removing id %p on RefBase %p"
                    "(weakref_type %p) that doesn't exist!",
                    id, mBase, this);
#endif

            ALOGE("RefBase: removing id %p on RefBase %p"
                    "(weakref_type %p) that doesn't exist!",
                    id, mBase, this);

            ref = head;
            while (ref) {
                char inc = ref->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, ref->id, ref->ref);
                ref = ref->next;
            }

            CallStack stack;
            stack.update();
            stack.dump();
        }
    }

    void renameRefsId(ref_entry *r, const void *old_id, const void *new_id)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);
            ref_entry *ref = r;
            while (ref != NULL) {
                if (ref->id == old_id) {
                    ref->id = new_id;
                }
                ref = ref->next;
            }
        }
    }

    void printRefsLocked(String *out, const ref_entry *refs) const
    {
        char buf[128];
```

```
        while (refs) {
            char inc = refs->ref >= 0 ? '+' : '-';
            sprintf(buf, "\t%c ID %p (ref %d):\n",
                    inc, refs->id, refs->ref);
            out->append(buf);
#if DEBUG_REFS_CALLSTACK_ENABLED
            out->append(refs->stack.toString("\t\t"));
#else
            out->append("\t\t(call stacks disabled)");
#endif
            refs = refs->next;
        }
    }

    mutable Mutex mMutex;
    ref_entry *mStrongRefs;
    ref_entry *mWeakRefs;

    bool mTrackEnabled;
    // Collect stack traces on addref and removeref,
    // instead of deleting the stack references
    // on removeref that match the address ones.
    bool mRetain;

#endif
};

// ---------------------------------------------------------------------------

void RefBase::incStrong(const void *id) const
{
    weakref_impl *const refs = mRefs;
    refs->incWeak(id);

    refs->addStrongRef(id);
    const int32_t c = android_atomic_inc(&refs->mStrong);
    ALOG_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);
#if PRINT_REFS
    ALOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);
#endif
    if (c != INITIAL_STRONG_VALUE)  {
        return;
    }

    android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
    refs->mBase->onFirstRef();
}

void RefBase::decStrong(const void *id) const
```

```
{
    weakref_impl *const refs = mRefs;
    refs->removeStrongRef(id);
    const int32_t c = android_atomic_dec(&refs->mStrong);
#if PRINT_REFS
    ALOGD("decStrong of %p from %p: cnt=%d\n", this, id, c);
#endif
    ALOG_ASSERT(c >= 1, "decStrong() called on %p too many times", refs);
    if (c == 1) {
        refs->mBase->onLastStrongRef(id);
        if ((refs->mFlags&OBJECT_LIFETIME_MASK) == OBJECT_LIFETIME_STRONG) {
            delete this;
        }
    }
    refs->decWeak(id);
}

void RefBase::forceIncStrong(const void *id) const
{
    weakref_impl* const refs = mRefs;
    refs->incWeak(id);

    refs->addStrongRef(id);
    const int32_t c = android_atomic_inc(&refs->mStrong);
    ALOG_ASSERT(c >= 0, "forceIncStrong called on %p after ref count underflow",
              refs);
#if PRINT_REFS
    ALOGD("forceIncStrong of %p from %p: cnt=%d\n", this, id, c);
#endif

    switch (c) {
    case INITIAL_STRONG_VALUE:
        android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
        // fall through...
    case 0:
        refs->mBase->onFirstRef();
    }
}

int32_t RefBase::getStrongCount() const
{
    return mRefs->mStrong;
}

RefBase* RefBase::weakref_type::refBase() const
{
    return static_cast<const weakref_impl*>(this)->mBase;
}
```

```
void RefBase::weakref_type::incWeak(const void *id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->addWeakRef(id);
    const int32_t c = android_atomic_inc(&impl->mWeak);
    ALOG_ASSERT(c >= 0, "incWeak called on %p after last weak ref", this);
}

void RefBase::weakref_type::decWeak(const void *id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id);
    const int32_t c = android_atomic_dec(&impl->mWeak);
    ALOG_ASSERT(c >= 1, "decWeak called on %p too many times", this);
    if (c != 1) return;

    if ((impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_STRONG) {
        // This is the regular lifetime case. The object is destroyed
        // when the last strong reference goes away. Since weakref_impl
        // outlive the object, it is not destroyed in the dtor, and
        // we'll have to do it here.
        if (impl->mStrong == INITIAL_STRONG_VALUE) {
            // Special case: we never had a strong reference, so we need to
            // destroy the object now.
            delete impl->mBase;
        } else {
            // ALOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
            delete impl;
        }
    } else {
        // less common case: lifetime is OBJECT_LIFETIME_{WEAK|FOREVER}
        impl->mBase->onLastWeakRef(id);
        if ((impl->mFlags&OBJECT_LIFETIME_MASK) == OBJECT_LIFETIME_WEAK) {
            // this is the OBJECT_LIFETIME_WEAK case. The last weak-reference
            // is gone, we can destroy the object.
            delete impl->mBase;
        }
    }
}

bool RefBase::weakref_type::attemptIncStrong(const void *id)
{
    incWeak(id);

    weakref_impl* const impl = static_cast<weakref_impl*>(this);

    int32_t curCount = impl->mStrong;
    ALOG_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
            this);
```

```
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
        curCount = impl->mStrong;
    }

    if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
        bool allow;
        if (curCount == INITIAL_STRONG_VALUE) {
            // Attempting to acquire first strong reference...  this is allowed
            // if the object does NOT have a longer lifetime (meaning the
            // implementation doesn't need to see this), or if the implementation
            // allows it to happen.
            allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
                  || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        } else {
            // Attempting to revive the object...  this is allowed
            // if the object DOES have a longer lifetime (so we can safely
            // call the object with only a weak ref) and the implementation
            // allows it to happen.
            allow = (impl->mFlags&OBJECT_LIFETIME_WEAK)==OBJECT_LIFETIME_WEAK
                  && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        }
        if (!allow) {
            decWeak(id);
            return false;
        }
        curCount = android_atomic_inc(&impl->mStrong);

        // If the strong reference count has already been incremented by
        // someone else, the implementor of onIncStrongAttempted() is holding
        // an unneeded reference.  So call onLastStrongRef() here to remove it.
        // (No, this is not pretty.)  Note that we MUST NOT do this if we
        // are in fact acquiring the first reference.
        if (curCount>0 && curCount<INITIAL_STRONG_VALUE) {
            impl->mBase->onLastStrongRef(id);
        }
    }

    impl->addStrongRef(id);

#if PRINT_REFS
    ALOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);
#endif

    if (curCount == INITIAL_STRONG_VALUE) {
        android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
        impl->mBase->onFirstRef();
```

```
    }

    return true;
}

bool RefBase::weakref_type::attemptIncWeak(const void *id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);

    int32_t curCount = impl->mWeak;
    ALOG_ASSERT(curCount >= 0, "attemptIncWeak called on %p after underflow",
             this);
    while (curCount > 0) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mWeak) == 0) {
            break;
        }
        curCount = impl->mWeak;
    }

    if (curCount > 0) {
        impl->addWeakRef(id);
    }

    return curCount > 0;
}

int32_t RefBase::weakref_type::getWeakCount() const
{
    return static_cast<const weakref_impl*>(this)->mWeak;
}

void RefBase::weakref_type::printRefs() const
{
    static_cast<const weakref_impl*>(this)->printRefs();
}

void RefBase::weakref_type::trackMe(bool enable, bool retain)
{
    static_cast<weakref_impl*>(this)->trackMe(enable, retain);
}

RefBase::weakref_type* RefBase::createWeak(const void* id) const
{
    mRefs->incWeak(id);
    return mRefs;
}

RefBase::weakref_type* RefBase::getWeakRefs() const
{
```

```
    return mRefs;
}

RefBase::RefBase()
    : mRefs(new weakref_impl(this))
{
}

RefBase::~RefBase()
{
    if (mRefs->mStrong == INITIAL_STRONG_VALUE) {
        // we never acquired a strong (and/or weak) reference on this object.
        delete mRefs;
    } else {
        // life-time of this object is extended to WEAK or FOREVER, in
        // which case weakref_impl doesn't out-live the object and we
        // can free it now.
        if ((mRefs->mFlags & OBJECT_LIFETIME_MASK) != OBJECT_LIFETIME_STRONG) {
            // It's possible that the weak count is not 0 if the object
            // re-acquired a weak reference in its destructor
            if (mRefs->mWeak == 0) {
                delete mRefs;
            }
        }
    }
    // for debugging purposes, clear this.
    const_cast<weakref_impl*&>(mRefs) = NULL;
}

void RefBase::extendObjectLifetime(int32_t mode)
{
    android_atomic_or(mode, &mRefs->mFlags);
}

void RefBase::onFirstRef()
{
}

void RefBase::onLastStrongRef(const void* /*id*/)
{
}

bool RefBase::onIncStrongAttempted(uint32_t flags, const void* id)
{
    return (flags&FIRST_INC_STRONG) ? true : false;
}

void RefBase::onLastWeakRef(const void* /*id*/)
{
```

```
}

// ----------------------------------------------------------------------

void RefBase::moveReferences(void *dst, void const *src, size_t n,
      const ReferenceConverterBase &caster)
{
#if DEBUG_REFS
   const size_t itemSize = caster.getReferenceTypeSize();
   for (size_t i=0; i<n; i++) {
      void *d = reinterpret_cast<void*>(intptr_t(dst) + i*itemSize);
      void const *s = reinterpret_cast<void const*>(intptr_t(src) + i*itemSize);
      RefBase* ref(reinterpret_cast<RefBase*>(caster.getReferenceBase(d)));
      ref->mRefs->renameStrongRefId(s, d);
      ref->mRefs->renameWeakRefId(s, d);
   }
#endif
}

//----------------------------------------------------------------------
TextOutput& printStrongPointer(TextOutput &to, const void *val)
{
   to << "sp<>(" << val << ")";
   return to;
}

TextOutput& printWeakPointer(TextOutput &to, const void *val)
{
   to << "wp<>(" << val << ")";
   return to;
}

}; // namespace android
```

整个上述代码被分成了如下所示的两大部分。

(1) 用如下 DEBUG_REFS 标记标识的部分，表示类 weakref_impl 被编译成调试版本。Debug 版本的源代码的成员函数都是有实现的，实现这些函数的目的，都是便于开发人员调试引用计数用：

```
#if !DEBUG_REFS
...
#else
```

(2) 用如下标记标识的部分，表示类 weakref_impl 被编译成非调试版本：

```
#else
...
#endif
```

4.5.5　弱指针

在 Android 系统中，弱指针和强指针使用一样的引用计数类：RefBase 类。与强指针类一样，弱指针也有一个指向目标对象的成员变量 m_ptr。另外，弱指针还有一个类型是 weakref_type 指针的额外的成员变量 m_refs。类 wp 在文件 frameworks\native\include\utils\RefBase.h 中定义，具体实现代码如下所示：

```
template <typename T>
class wp
{
public:
    typedef typename RefBase::weakref_type weakref_type;

    inline wp() : m_ptr(0) { }

    wp(T *other);
    wp(const wp<T> &other);
    wp(const sp<T> &other);
    template<typename U> wp(U *other);
    template<typename U> wp(const sp<U> &other);
    template<typename U> wp(const wp<U> &other);

    ~wp();

    // Assignment

    wp& operator = (T *other);
    wp& operator = (const wp<T> &other);
    wp& operator = (const sp<T> &other);

    template<typename U> wp& operator = (U *other);
    template<typename U> wp& operator = (const wp<U> &other);
    template<typename U> wp& operator = (const sp<U> &other);

    void set_object_and_refs(T *other, weakref_type *refs);

    // promotion to sp

    sp<T> promote() const;

    // Reset

    void clear();

    // Accessors

    inline  weakref_type* get_refs() const { return m_refs; }
```

```
    inline  T* unsafe_get() const { return m_ptr; }

    // Operators

    COMPARE_WEAK(==)
    COMPARE_WEAK(!=)
    COMPARE_WEAK(>)
    COMPARE_WEAK(<)
    COMPARE_WEAK(<=)
    COMPARE_WEAK(>=)

    inline bool operator == (const wp<T> &o) const {
        return (m_ptr == o.m_ptr) && (m_refs == o.m_refs);
    }
    template<typename U>
    inline bool operator == (const wp<U> &o) const {
        return m_ptr == o.m_ptr;
    }

    inline bool operator > (const wp<T> &o) const {
        return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
    }
    template<typename U>
    inline bool operator > (const wp<U> &o) const {
        return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
    }

    inline bool operator < (const wp<T> &o) const {
        return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
    }
    template<typename U>
    inline bool operator < (const wp<U> &o) const {
        return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
    }
    inline bool operator != (const wp<T> &o) const { return m_refs != o.m_refs; }
    template<typename U> inline bool operator != (const wp<U> &o) const
    { return !operator == (o); }
    inline bool operator <= (const wp<T> &o) const { return !operator > (o); }
    template<typename U> inline bool operator <= (const wp<U>& o) const
    { return !operator > (o); }
    inline bool operator >= (const wp<T> &o) const { return !operator < (o); }
    template<typename U> inline bool operator >= (const wp<U> &o) const
    { return !operator < (o); }

private:
    template<typename Y> friend class sp;
    template<typename Y> friend class wp;
```

```
    T *m_ptr;
    weakref_type *m_refs;
};
```

类 wp 的构造函数的实现代码如下所示：

```
template<typename T>
wp<T>::wp(T *other)
  : m_ptr(other)
{
    if (other) m_refs = other->createWeak(this);
}
```

在上述代码中，参数 other 类继承于类 RefBase，并调用了类 RefBase 的成员函数 createWeak。函数 createWeak 在文件 frameworks\native\libs\utils\RefBase.cpp 中定义，具体实现代码如下所示：

```
RefBase::weakref_type* RefBase::createWeak(const void *id) const
{
    mRefs->incWeak(id);
    return mRefs;  //mRefs 的类型为 weakref_impl 指针
}
```

再看类 wp 的析构函数，此函数直接调用目标对象的 weakref_impl 对象的函数 decWeak，目的是减少弱引用计数。当弱引用计数为 0 时，根据在目标对象的标志位(0、OBJECT_LIFETIME_WEAK 或者 OBJECT_LIFETIME_FOREVER)来决定是否要 delete(删除)目标对象。下面是析构函数的实现代码：

```
template<typename T>
wp<T>::~wp()
{
    if (m_ptr) m_refs->decWeak(this);
}
```

弱指针的最大特点是不能直接操作目标对象，原因是弱指针类没有重载“*”和“->”操作符，而强指针重载了这两个操作符。如果坚持要操作目标对象，则需要把弱指针升级为强指针。升级方法是使用成员变量 m_ptr 和 m_refs 构造一个强指针 sp，m_ptr 是指目标对象的一个指针，m_refs 指向目标对象里面的 weakref_impl 对象。升级代码如下：

```
template<typename T>
sp<T> wp<T>::promote() const
{
    return sp<T>(m_ptr, m_refs);
}
```

与之对应的强指针构造代码如下所示：

```
template<typename T>
sp<T>::sp(T *p, weakref_type *refs)
  : m_ptr((p && refs->attemptIncStrong(this))? p : 0)
```

```
{
}
```

在上述构造代码中，初始化指向了目标对象的成员变量 m_ptr。如果还存在目标对象，则 m_ptr 指向目标对象。如果这个目标对象已经不存在，则 m_ptr 为 NULL。是否升级成功需要参考函数 attemptIncStrong 的返回结果。函数 attemptIncStrong 的具体实现代码如下所示：

```
bool RefBase::weakref_type::attemptIncStrong(const void *id)
{
    incWeak(id);

    weakref_impl* const impl = static_cast<weakref_impl*>(this);

    int32_t curCount = impl->mStrong;
    LOG_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
        this);
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
        curCount = impl->mStrong;
    }

    if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
        bool allow;
        if (curCount == INITIAL_STRONG_VALUE) {
            // Attempting to acquire first strong reference...  this is allowed
            // if the object does NOT have a longer lifetime (meaning the
            // implementation doesn't need to see this), or if the implementation
            // allows it to happen.
            allow=(impl->mFlags&OBJECT_LIFETIME_WEAK)!=OBJECT_LIFETIME_WEAK
              || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        } else {
            // Attempting to revive the object...  this is allowed
            // if the object DOES have a longer lifetime (so we can safely
            // call the object with only a weak ref) and the implementation
            // allows it to happen.
            allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
                && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        }
        if (!allow) {
            decWeak(id);
            return false;
        }
        curCount = android_atomic_inc(&impl->mStrong);

        // If the strong reference count has already been incremented by
        // someone else, the implementor of onIncStrongAttempted() is holding
        // an unneeded reference.  So call onLastStrongRef() here to remove it.
```

```
        // (No, this is not pretty.)  Note that we MUST NOT do this if we
        // are in fact acquiring the first reference.
        if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
            impl->mBase->onLastStrongRef(id);
        }
    }

    impl->addWeakRef(id);
    impl->addStrongRef(id);

#if PRINT_REFS
    LOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);
#endif

    if (curCount == INITIAL_STRONG_VALUE) {
        android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
        impl->mBase->onFirstRef();
    }

    return true;
}
```

Android

第 5 章 Android虚拟机系统详解

Android 的虚拟机系统是 Dalvik VM(VM 是虚拟机的缩写)，是 Google 等商家合作开发的 Android 移动设备平台的核心组成部件之一。Dalvik VM 可以支持已转换为.dex(即 Dalvik Executable)格式的 Java 应用程序的运行，其中“.dex”格式是专为 Dalvik 设计的一种压缩格式，适合内存和处理器速度有限的系统。现实中的大多数虚拟机都是一种堆栈机器，例如 JVM(Java 的虚拟机)，而 Dalvik 虚拟机则是基于寄存器的。基于栈的机器需要更多指令，而基于寄存器的机器指令更大。本章将详细分析 Dalvik VM 系统的源码结构，为读者步入本书后面知识的学习打下基础。

5.1 Android虚拟机基础

Android 系统的应用层是采用 Java 开发的，由于 Java 语言的跨平台特性，所以 Java 代码必须运行在虚拟机中。正是因为这个特性，Android 系统也实现了自己的一个类似 JVM 但是更适合于嵌入式平台的 Java 虚拟机，这被称为 Dalvik。Dalvik 的功能等同于 JVM，为 Android 平台上的 Java 代码提供了运行环境。

5.1.1 Android虚拟机源码目录

当下载好 Android 4.3 的源码后，因为 Dalvik 本身是由 C++语言实现的，在源码中，根目录下有 dalvik 文件夹，里面存放的是 dalvik 虚拟机的实现代码，其目录结构如下所示：

```
./
├── dalvikvm                    //入口目录
├── dexdump                     //dex 反汇编
├── dexgen                      //dex 生成相关
├── dexlist                     //dex 列表
├── dexopt                      //验证和优化
├── docs                        //文档
├── dvz                         //zygot 相关
├── dx                          //dx 工具，将多个 Java 转换为 dex
├── hit
├── libdex                      //dex 库的实现代码
├── opcode-gen
├── tests                       //测试相关
├── tools                       //工具
├── unit-tests                  //测试相关
├── vm                          //虚拟机的实现
├── Android.mk                  //Makefile
├── CleanSpec.mk
├── MODULE_LICENSE_APACHE2
├── NOTICE
└── README.txt
```

dalvik/目录的结构如图 5-1 所示。

Dalvik 虚拟机目录结构的具体说明如下所示。

- android.mk：是虚拟机编译的 makefile 文件。
- dalvikvm：此目录包含虚拟机命令行调用入口文件，主要用来解释命令行参数，调用库函数接口等。
- dexdump：此目录包含 dex 文件的反编译查看工具，主要用来查看编译出来的代码文件是否正确，查看编译出来的文件结构如何。
- dexlist：此目录包含查看 dex 文件里所有类的方法的工具。
- dexopt：此目录包含 dex 优化工具。

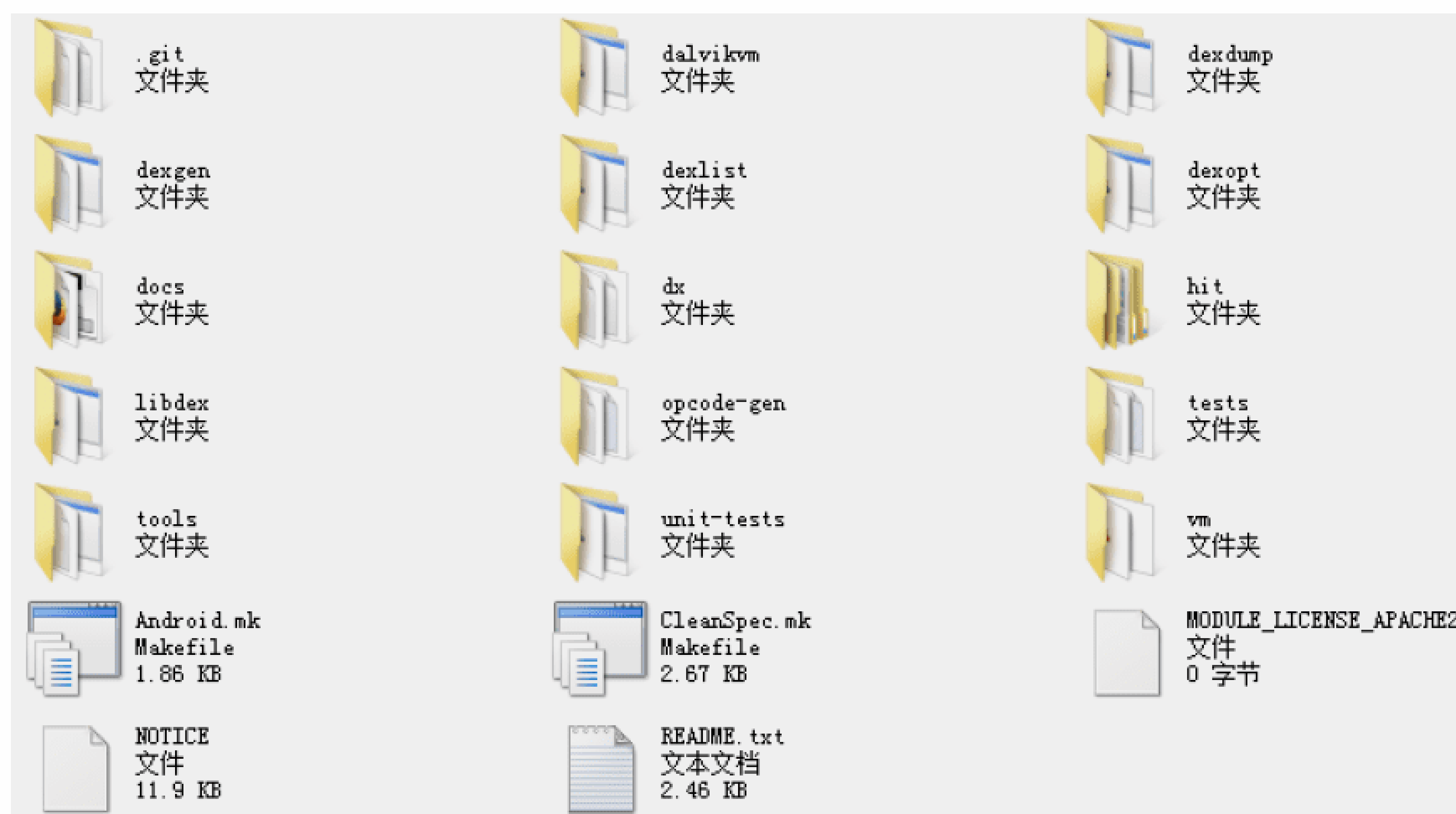


图 5-1　dalvik/目录的结构

- docs：此目录用来保存 Dalvik 虚拟机相关的帮助文档。
- dvz：此目录包含从 zygote 请求生成虚拟机实例的工具。
- dx：此目录包含从 Java 字节码转换为 Dalvik 机器码的工具。
- hit：此目录包含显示堆栈信息/对象信息的工具。
- libcore：此目录包含 Dalvik 虚拟机的核心类库，提供给上层的应用程序调用。
- libcore-disabled：此目录包含一些禁用的库。
- libdex：此目录包含主机和设备处理 dex 文件的库。
- libnativehelper：此目录包含 Dalvik 虚拟核心库的支持库函数。
- MODULE_LICENSE_APACHE2：这是 Apache 2 的版权声明文件。
- NOTICE：该文件用来说明虚拟机源码的版权注意事项。
- README.txt：该文件用来说明目录相关内容和版权。
- run-core-tests.sh：该文件用来运行核心库测试。
- tests：此目录包含测试相关的测试用例。
- tools：此目录包含一些编译/运行相关的工具。
- vm：此目录包含虚拟机的绝大部分代码，包括指令读取、指令执行等。

正是因为 Android 虚拟机有上述实现代码，所以应用程序生成的二进制执行文件能够快速、稳定地运行在 Android 系统上。

5.1.2　Dalvik的架构

在 Android 源码中，Dalvik 虚拟机的实现位于 dalvik/目录下，其中 dalvik/vm 是虚拟机的实现部分，将会编译成 libdvm.so。而 dalvik/libdex 将会编译成 libdex.a 静态库，dalvik/dexdump 是.dex 文件的反编译工具，虚拟机的可执行程序位于 dalvik/dalvikvm 中，将会编译成 dalvikvm 可执行文件。

Dalvik 虚拟机的架构如图 5-2 所示。

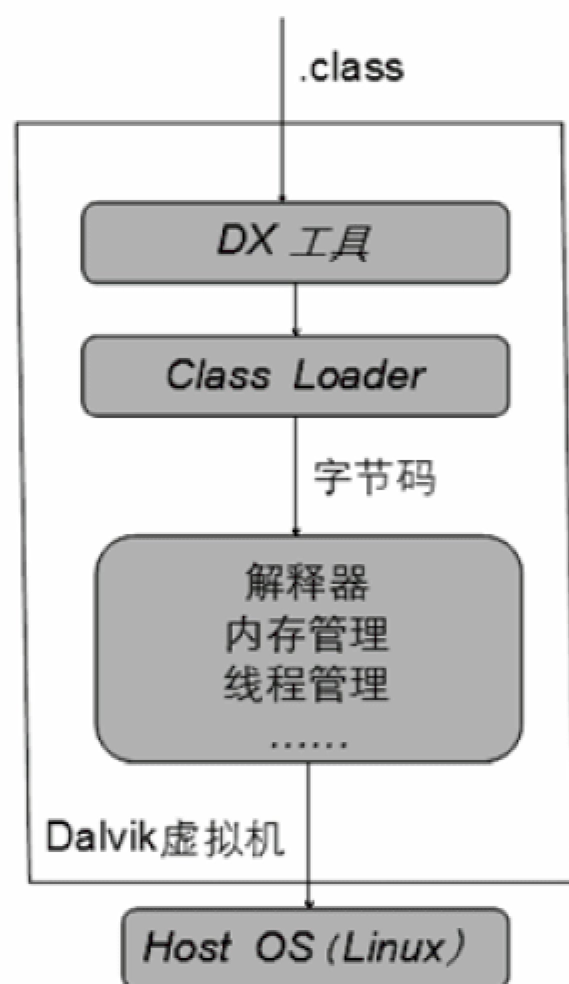


图 5-2 Dalvik虚拟机的架构

Android 应用的编译及运行流程如图 5-3 所示。

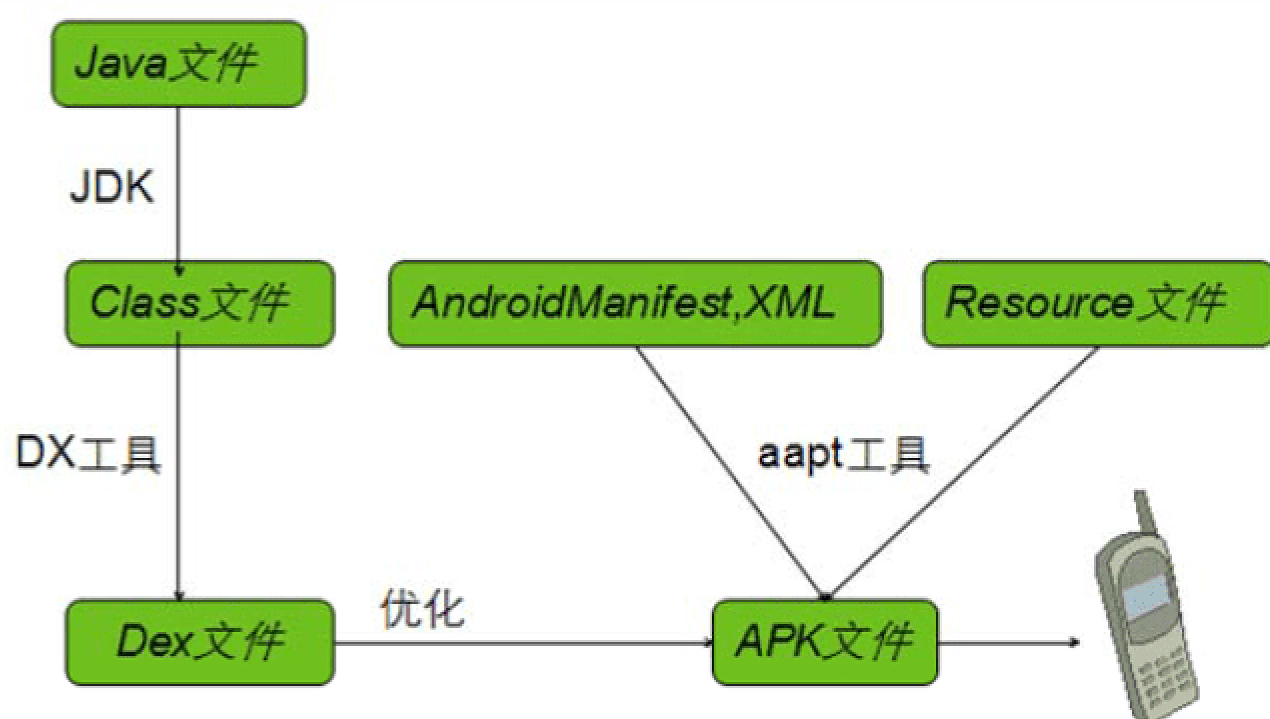


图 5-3 Android应用的编译及运行流程

DX 工具用来转换 Java class 成为 DEX 格式，但不是全部。多个类型包含在一个 Dex 文件中。多个类型中重复的字符串和其他常数在 DEX 中只存放一次，以节省空间。Java 字节码(bytecode)转换成 Dalvik 虚拟机所使用的替代指令集。一个未压缩 Dex 文件通常是稍稍小于一个已经压缩的 JAR 文件。

再次安装到执行设备时，Dalvik 可执行文件可能会是修改过的。为了获得进一步的优化，端序(Byte Order)可能会存在一定的数据交换，简单的数据结构和函数库可内联(Linked Inline)，空的类型对象可能会短路处理。

当启动 Android 时，Dalvik VM 会监视所有的程序(APK)，并且创建依存关系树，为每个程序优化代码并存储在 Dalvik 缓存中。Dalvik 第一次加载后会生成 Cache 文件，以提供下次快速加载，所以第一次会比较慢。

Dalvik 解释器采用预先算好的 Goto 地址，基于每个指令集 OpCode，都固定以 64 字节为 Memory Alignment。这样可以节省一个指令集 OpCode 后要进行查表的时间。为了强化功能，

Dalvik 还提供了 Fast Interpreter(快速翻译器)。

DX 是一套工具，可以将 Java 的.class 文件转换成.dex 格式。一个 Dex 文件通常会有多个.class 文件。Dex 有时必须进行优化，这会使文件大小增加 1~4 倍，以 ODEX 结尾。

5.1.3　Dalvik虚拟机的主要特征

在 Dalvik 虚拟机中，一个应用中会定义很多类，编译完成后，即会有很多相应的 Class 文件，Class 文件间会有不少冗余的信息；而 Dex 文件格式会把所有的 Class 文件内容整合到一个文件中。这样，除了可以减少整体的文件尺寸、I/O 操作，也提高了类的查找速度。原来每个类文件中的常量池，在 Dex 文件中由一个常量池来管理。

每一个 Android 应用都运行在一个 Dalvik 虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制、内存分配和管理、Mutex 等都是依赖底层操作系统实现的。所有 Android 应用的线程都对应一个 Linux 线程，虚拟机因而可以更多地依赖操作系统的线程调度和管理机制。

不同的应用在不同的进程空间里运行，加之对不同来源的应用都使用不同的 Linux 用户来运行，这可以最大程度地保护应用的安全和独立运行。

Zygote 是一个虚拟机进程，同时也是一个虚拟机实例的孕育器，每当系统要求执行一个 Android 应用程序时，Zygote 就会 Fork 出一个子进程来执行该应用程序。这样做的好处显而易见：Zygote 进程是在系统启动时产生的，它会完成虚拟机的初始化、库的加载、预置类库的加载和初始化等操作，而在系统需要一个新的虚拟机实例时，Zygote 通过复制自身，最快速地提供一个系统。另外，对于一些只读的系统库，所有虚拟机实例都与 Zygote 共享一块内存区域，大大节省了内存开销。

相对于基于堆栈的虚拟机实现，基于寄存器的虚拟机实现虽然在硬件通用性上要差一些，但是它在代码的执行效率上却更胜一筹。在基于寄存器的虚拟机里，可以更为有效地减少冗余指令的分发和减少内存的读写访问。

5.1.4　Dalvik的进程管理

Dalvik 进程管理是依赖于 Linux 的进程体系结构的，如要为应用程序创建一个进程，它会使用 Linux 的 Fork 机制来复制一个进程(复制进程往往比创建进程效率更高)。

Zygote 通过 Init 进程启动。首先会孕育出 System_Server(Android 绝大多系统服务的守护进程，它会监听 socket，等待请求命令，当有一个应用程序启动时，就会向它发出请求，Zygote 就会 Fork 出一个新的应用程序进程)。每当系统要求执行一个 Android 应用程序时，Zygote 就会运用 Linux 的 Fork 机制产生一个子进程来执行该应用程序。

5.1.5　Android 的初始化流程

Linux 中进程间通信的方式很多，但 Dalvik 使用的是信号方式来完成进程间通信。Android 的初始化流程如图 5-4 所示。

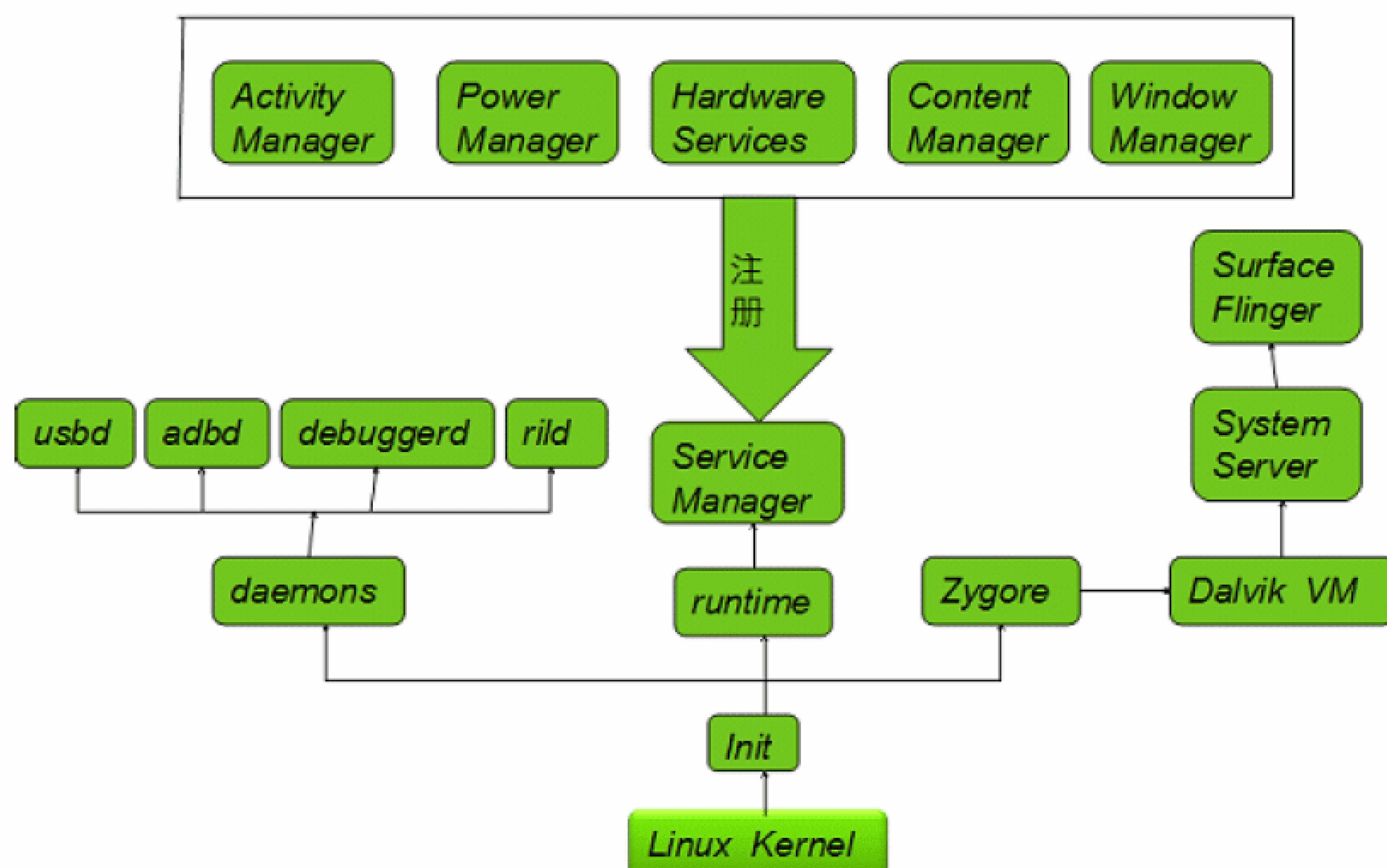


图 5-4　Android的初始化流程

5.2　分析Dalvik的运作流程

经过本章前面内容的学习，已经了解了 Android 虚拟机 Dalvik 的基本知识。在本节的内容中，将详细讲解 Dalvik 虚拟机的运作流程，为读者步入本书后面知识的学习打下基础。

5.2.1　Dalvik虚拟机相关的可执行程序

在 Android 源码中，我们会发现好几处与 Dalvik 这个概念相关的可执行程序，正确区分这些可执行程序的区别，将有助于理解 Framework 内部结构。这些可执行程序的名称和源码路径如表 5-1 所示。

表 5-1　与虚拟机相关的源码

名　称	源码路径
dalvikvm	dalvik/dalvikvm
dvz	dalvik/dvz
app_process	frameworks/base/cmds/app_process

在接下来的内容中，将分别介绍这些可执行程序的作用。

1. dalvikvm

当运行 Java 程序时，都是由一个虚拟机来解释 Java 的字节码，它将这些字节码翻译成本地 CPU 的指令码，然后执行。对 Java 程序来说，负责解释并执行的就是一个虚拟机。而对于 Linux 系统而言，这个进程只是一个普通的进程，它与一个只有一行代码的 Hello World 可执行程序无本质区别。所以启动一个虚拟机的方法就跟启动任何一个可执行程序的方法是相同的，

那就是在命令行下输入可执行程序的名称，并在参数中指定要执行的 Java 类。dalvikvm 的执行语法如下：

```
dalvikvm -cp 类路径 类名
```

由此可以看到，dalvikvm 的作用就像在 PC 上执行 Java 程序一样。

2. dvz

在 Dalvik 虚拟机中，dvz 的作用是从 Zygote 进程中孕育出一个新的进程，新的进程也是一个 Dalvik 虚拟机。该进程与 dalvikvm 启动的虚拟机相比，区别在于该进程中已经预装了 Framework 的大部分类和资源。使用 dvz 的语法格式如下：

```
dvz  -classpath 包名称 类名
```

一个 APK 的入口类是 ActivityThread 类。因为 Activity 类仅仅是被回调的类，所以不可以通过 Activity 类来启动一个 APK，dvz 工具仅仅用于 Framework 开发过程的调试阶段。

3. app_process

前面讲解的 dalvikvm 和 dvz 是两个通用的工具，然而 Framework 在启动时需要加载并运行如下两个特定 Java 类(文件)：ZygoteInit.java 和 SystemServer.java。

为了便于使用，系统才提供了一个 app_process 进程，该进程会自动运行这两个类，从这个角度来讲，app_process 的本质就是使用 dalvikvm 来启动 ZygoteInit.java，并在启动后加载 Framework 中的大部分类和资源。

在接下来的内容中，我们将对比 app_process 和 dalvikvm 的主要执行过程。

(1)　首先看 dalvikvm，其源码在文件 dalvik/dalvikvm/Main.c 中，关键代码有两处：

```
/*
 * 第一处：通过如下代码创建一个 vm 对象
 */
if (JNI_CreateJavaVM(&vm, &env, &initArgs) < 0) {
    fprintf(stderr, "Dalvik VM init failed (check log file)\n");
    goto bail;
}

/*
 * Make sure they provided a class name.  We do this after VM init
 * so that things like "-Xrunjdwp:help" have the opportunity to emit
 * a usage statement.
 */
if (argIdx == argc) {
    fprintf(stderr, "Dalvik VM requires a class name\n");
    goto bail;
}

/*
 * We want to call main() with a String array with our arguments in it.
 * Create an array and populate it.  Note argv[0] is not included.
```

```
 */
jobjectArray strArray;
strArray = createStringArray(env, &argv[argIdx+1], argc-argIdx-1);
if (strArray == NULL)
    goto bail;

/*
 * Find [class].main(String[]).
 */
jclass startClass;
jmethodID startMeth;
char *cp;

/* convert "com.android.Blah" to "com/android/Blah" */
slashClass = strdup(argv[argIdx]);
for (cp = slashClass; *cp != '\0'; cp++)
    if (*cp == '.')
        *cp = '/';
/*第二处：创建好了 JavaVm 对象后，就可以使用该对象去加载指定的类了*/
startClass = (*env)->FindClass(env, slashClass);
if (startClass == NULL) {
    fprintf(stderr, "Dalvik VM unable to locate class '%s'\n", slashClass);
    goto bail;
}

startMeth = (*env)->GetStaticMethodID(env, startClass,
                "main", "([Ljava/lang/String;)V");
if (startMeth == NULL) {
    fprintf(stderr, "Dalvik VM unable to find static main(String[]) in '%s'\n",
        slashClass);
    goto bail;
}

/*
 * Make sure the method is public.  JNI doesn't prevent us from calling
 * a private method, so we have to check it explicitly.
 */
if (!methodIsPublic(env, startClass, startMeth))
    goto bail;

/*
 * Invoke main().
 */
(*env)->CallStaticVoidMethod(env, startClass, startMeth, strArray);

if (!(*env)->ExceptionCheck(env))
    result = 0;
```

在上述第一处关键代码处，该段代码调用 JNI_CreateJavaVM()，并同时创建了 JavaVm 对

象和 JNIEnv 对象，这两个对象的定义如下：

```
JNIEnvExt *pEnv = NULL;
JavaVMExt *pVM = NULL;
```

该函数的参数是“指针的指针”类型，其原型如下所示：

```
jint JNI_CreateJavaVM(JavaVM **p_vm, JNIEnv **p_env, void *vm_args)
```

在上述第二处关键代码处，首先调用 FindClass()找到指定的 class 文件，然后调用 GetStaticMethodID()找到 main()函数，最后调用 CallStaticVoidMethod 执行该 main()函数。

(2) 接下来看 app_process 中是如何创建虚拟机并执行指定的 class 文件的。其源代码在文件 frameworks/base/cmds/app_main.cpp 中，该文件中的关键代码有如下两处。

- 第一处：先创建一个 AppRuntime 对象。
- 第二处：调用 runtime 的 start()方法启动指定的 class。

在系统中，只有一处使用了 app_process，那就是在 init.rc 中。因为在使用时参数包含了“--zygote”及“--start-system-server”，所以这里仅分析包含这两个参数的情况。

start()方式是类 AppRuntime 的成员函数，而 AppRuntime 是在该文件中定义的一个应用类，其父类是 AndroidRuntime，该类的实现在文件 frameworkds/base/core/jni/AndroidRuntime.cpp 中。在函数 start()中，首先调用 startVm()创了建一个 vm 对象，然后就和 dalvikvm 一样先找到 Class()，再执行 class 中的 main()函数，使用 startVm()函数创建 vm 对象。

由上述过程可以看出，app_process 和 dalvikvm 在本质上是相同的，唯一的区别就是 app_process 可以指定一些特别的参数，这些参数有利于 Framework 启动特定的类，并进行一些特别的系统环境参数设置。

5.2.2　初始化Dalvik虚拟机

在 Dalvik 虚拟机运行伊始，先进行的是初始化工作，此工作的核心实现文件是 Init.c。在本节的内容中，将详细讲解 Dalvik 虚拟机的初始化过程。

1. 开始虚拟机的准备工作

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmStartup()实现所有启动虚拟机的准备工作，此函数的具体实现代码如下所示：

```
int dvmStartup(int argc, const char* const argv[], bool ignoreUnrecognized,
  JNIEnv *pEnv)
{
    int i, cc;
    assert(gDvm.initializing);
    LOGV("VM init args (%d):\n", argc);
    for (i=0; i<argc; i++)
        LOGV("  %d: '%s'\n", i, argv[i]);
    setCommandLineDefaults();
    /* prep properties storage */
    if (!dvmPropertiesStartup(argc))
        goto fail;
```

```
    /*
     * Process the option flags (if any).
     */
    cc = dvmProcessOptions(argc, argv, ignoreUnrecognized);
    if (cc != 0) {
        if (cc < 0) {
            dvmFprintf(stderr, "\n");
            dvmUsage("dalvikvm");
        }
        goto fail;
    }
}
```

2. 初始化跟踪显示系统

在Dalvik虚拟机的初始化过程中，使用函数dvmAllocTrackerStartup()初始化跟踪显示系统，跟踪系统主要用来生成调试系统的数据包。此函数的具体实现代码如下所示：

```
bool dvmAllocTrackerStartup(void)
{
    /* prep locks */
    dvmInitMutex(&gDvm.allocTrackerLock);
    /* initialized when enabled by DDMS */
    assert(gDvm.allocRecords == NULL);
    return true;
}
```

上述函数的实现代码保存在文件 AllocTracker.c 中。

3. 初始化垃圾回收器

在 Dalvik 虚拟机的初始化过程中，使用 dvmGcStartup()函数初始化垃圾回收器，此函数的具体实现代码如下所示：

```
bool dvmGcStartup(void)
{
    dvmInitMutex(&gDvm.gcHeapLock);
    return dvmHeapStartup();
}
```

4. 初始化线程列表和主线程环境参数

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmThreadStartup()初始化线程列表和主线程环境参数，此函数的具体实现代码如下所示：

```
bool dvmThreadStartup(void)
{
    Thread *thread;

    /* allocate a TLS slot */
```

```
    if (pthread_key_create(&gDvm.pthreadKeySelf, threadExitCheck) != 0) {
        LOGE("ERROR: pthread key create failed\n");
        return false;
    }

    /* test our pthread lib */
    if (pthread_getspecific(gDvm.pthreadKeySelf) != NULL)
        LOGW("WARNING: newly-created pthread TLS slot is not NULL\n");

    /* prep thread-related locks and conditions */
    dvmInitMutex(&gDvm.threadListLock);
    pthread_cond_init(&gDvm.threadStartCond, NULL);
    //dvmInitMutex(&gDvm.vmExitLock);
    pthread_cond_init(&gDvm.vmExitCond, NULL);
    dvmInitMutex(&gDvm._threadSuspendLock);
    dvmInitMutex(&gDvm.threadSuspendCountLock);
    pthread_cond_init(&gDvm.threadSuspendCountCond, NULL);
#ifdef WITH_DEADLOCK_PREDICTION
    dvmInitMutex(&gDvm.deadlockHistoryLock);
#endif

    /*
     * Dedicated monitor for Thread.sleep().
     * TODO: change this to an Object* so we don't have to expose this
     * call, and we interact better with JDWP monitor calls.  Requires
     * deferring the object creation to much later (e.g. final "main"
     * thread prep) or until first use.
     */
    gDvm.threadSleepMon = dvmCreateMonitor(NULL);

    gDvm.threadIdMap = dvmAllocBitVector(kMaxThreadId, false);

    thread = allocThread(gDvm.stackSize);
    if (thread == NULL)
        return false;

    /* switch mode for when we run initializers */
    thread->status = THREAD_RUNNING;

    /*
     * We need to assign the threadId early so we can lock/notify
     * object monitors.  We'll set the "threadObj" field later.
     */
    prepareThread(thread);
    gDvm.threadList = thread;

#ifdef COUNT_PRECISE_METHODS
    gDvm.preciseMethods = dvmPointerSetAlloc(200);
#endif
```

```
    return true;
}
```

上述函数的实现代码保存在文件 Thread.c 中。

5. 分配内部操作方法的表格内存

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmInlineNativeStartup()分配内部操作方法的表格内存，此函数的具体实现代码如下所示：

```
bool dvmInlineNativeStartup(void)
{
#ifdef WITH PROFILER
    gDvm.inlinedMethods =
        (Method**) calloc(NELEM(gDvmInlineOpsTable), sizeof(Method*));
    if (gDvm.inlinedMethods == NULL)
        return false;
#endif
    return true;
}
```

上述函数的实现代码保存在文件 InlineNative.c 中。

6. 初始化虚拟机的指令码相关的内容

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmVerificationStartup()初始化虚拟机的指令码相关的内容，以便检查指令是否正确。此函数的具体实现代码如下所示：

```
bool dvmVerificationStartup(void)
{
    gDvm.instrWidth = dexCreateInstrWidthTable();
    gDvm.instrFormat = dexCreateInstrFormatTable();
    gDvm.instrFlags = dexCreateInstrFlagsTable();
    if (gDvm.instrWidth == NULL || gDvm.instrFormat == NULL
      || gDvm.instrFlags == NULL)
    {
        LOGE("Unable to create instruction tables\n");
        return false;
    }
    return true;
}
```

上述函数的实现代码保存在文件 analysis\DexVerify.c 中。

7. 分配指令寄存器状态的内存

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmRegisterMapStartup()分配指令寄存器状态的内存。此函数的具体实现代码如下所示：

```
bool dvmRegisterMapStartup(void)
```

```
{
#ifdef REGISTER_MAP_STATS
    MapStats *pStats = calloc(1, sizeof(MapStats));
    gDvm.registerMapStats = pStats;
#endif
    return true;
}
```

上述函数的实现代码保存在文件 analysis\RegisterMap.c 中。

8. 分配指令寄存器状态的内存

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmInstanceofStartup()分配虚拟机使用的缓存。此函数的具体实现代码如下所示：

```
bool dvmInstanceofStartup(void)
{
    gDvm.instanceofCache = dvmAllocAtomicCache(INSTANCEOF_CACHE_SIZE);
    if (gDvm.instanceofCache == NULL)
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 oo\TypeCheck.c 中。

9. 初始化虚拟机用的最基本Java库

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmClassStartup()初始化虚拟机用的最基本 Java 库。此函数的具体实现代码如下所示：

```
bool dvmClassStartup(void)
{
    ClassObject *unlinkedClass;
    /* make this a requirement -- don't currently support dirs in path */
    if (strcmp(gDvm.bootClassPathStr, ".") == 0) {
        LOGE("ERROR: must specify non-'.' bootclasspath\n");
        return false;
    }
    gDvm.loadedClasses =
        dvmHashTableCreate(256, (HashFreeFunc)dvmFreeClassInnards);
    gDvm.pBootLoaderAlloc = dvmLinearAllocCreate(NULL);
    if (gDvm.pBootLoaderAlloc == NULL)
        return false;
    if (false) {
        linearAllocTests();
        exit(0);
    }
    /*
     * Class serial number.  We start with a high value to make it distinct
     * in binary dumps (e.g. hprof).
     */
```

```
    gDvm.classSerialNumber = INITIAL_CLASS_SERIAL_NUMBER;
    /* Set up the table we'll use for tracking initiating loaders for
     * early classes.
     * If it's NULL, we just fall back to the InitiatingLoaderList in the
     * ClassObject, so it's not fatal to fail this allocation.
     */
    gDvm.initiatingLoaderList =
        calloc(ZYGOTE_CLASS_CUTOFF, sizeof(InitiatingLoaderList));
    /* This placeholder class is used while a ClassObject is
     * loading/linking so those not in the know can still say
     * "obj->clazz->...".
     */
    unlinkedClass = &gDvm.unlinkedJavaLangClassObject;
    memset(unlinkedClass, 0, sizeof(*unlinkedClass));
    /* Set obj->clazz to NULL so anyone who gets too interested
     * in the fake class will crash.
     */
    DVM_OBJECT_INIT(&unlinkedClass->obj, NULL);
    unlinkedClass->descriptor = "!unlinkedClass";
    dvmSetClassSerialNumber(unlinkedClass);
    gDvm.unlinkedJavaLangClass = unlinkedClass;
    /*
     * Process the bootstrap class path.  This means opening the specified
     * DEX or Jar files and possibly running them through the optimizer.
     */
    assert(gDvm.bootClassPath == NULL);
    processClassPath(gDvm.bootClassPathStr, true);
    if (gDvm.bootClassPath == NULL)
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 oo\Class.c 中。

10. 使用的Java类库线程类

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmThreadObjStartup()初始化虚拟机进一步使用的 Java 类库线程类。此函数的具体实现代码如下所示：

```
bool dvmThreadObjStartup(void)
{
    /*
     * Cache the locations of these classes.  It's likely that we're the
     * first to reference them, so they're being loaded now.
     */
    gDvm.classJavaLangThread =
        dvmFindSystemClassNoInit("Ljava/lang/Thread;");
    gDvm.classJavaLangVMThread =
        dvmFindSystemClassNoInit("Ljava/lang/VMThread;");
    gDvm.classJavaLangThreadGroup =
```

```
        dvmFindSystemClassNoInit("Ljava/lang/ThreadGroup;");
    if (gDvm.classJavaLangThread == NULL
      || gDvm.classJavaLangThreadGroup == NULL
      || gDvm.classJavaLangThreadGroup == NULL)
    {
        LOGE("Could not find one or more essential thread classes\n");
        return false;
    }

    /*
     * Cache field offsets.  This makes things a little faster, at the
     * expense of hard-coding non-public field names into the VM.
     */
    gDvm.offJavaLangThread_vmThread =
        dvmFindFieldOffset(gDvm.classJavaLangThread,
            "vmThread", "Ljava/lang/VMThread;");
    gDvm.offJavaLangThread_group =
        dvmFindFieldOffset(gDvm.classJavaLangThread,
            "group", "Ljava/lang/ThreadGroup;");
    gDvm.offJavaLangThread_daemon =
        dvmFindFieldOffset(gDvm.classJavaLangThread, "daemon", "Z");
    gDvm.offJavaLangThread_name =
        dvmFindFieldOffset(gDvm.classJavaLangThread,
            "name", "Ljava/lang/String;");
    gDvm.offJavaLangThread_priority =
        dvmFindFieldOffset(gDvm.classJavaLangThread, "priority", "I");

    if (gDvm.offJavaLangThread_vmThread < 0
      || gDvm.offJavaLangThread_group < 0
      || gDvm.offJavaLangThread_daemon < 0
      || gDvm.offJavaLangThread_name < 0
      || gDvm.offJavaLangThread_priority < 0)
    {
        LOGE("Unable to find all fields in java.lang.Thread\n");
        return false;
    }

    gDvm.offJavaLangVMThread_thread =
        dvmFindFieldOffset(gDvm.classJavaLangVMThread,
            "thread", "Ljava/lang/Thread;");
    gDvm.offJavaLangVMThread_vmData =
        dvmFindFieldOffset(gDvm.classJavaLangVMThread, "vmData", "I");
    if (gDvm.offJavaLangVMThread_thread < 0
      || gDvm.offJavaLangVMThread_vmData < 0)
    {
        LOGE("Unable to find all fields in java.lang.VMThread\n");
        return false;
    }
```

```
    /*
     * Cache the vtable offset for "run()".
     *
     * We don't want to keep the Method* because then we won't find see
     * methods defined in subclasses.
     */
    Method *meth;
    meth =
      dvmFindVirtualMethodByDescriptor(gDvm.classJavaLangThread, "run", "()V");
    if (meth == NULL) {
        LOGE("Unable to find run() in java.lang.Thread\n");
        return false;
    }
    gDvm.voffJavaLangThread_run = meth->methodIndex;

    /*
     * Cache vtable offsets for ThreadGroup methods.
     */
    meth = dvmFindVirtualMethodByDescriptor(gDvm.classJavaLangThreadGroup,
        "removeThread", "(Ljava/lang/Thread;)V");
    if (meth == NULL) {
        LOGE("Unable to find removeThread(Thread) in java.lang.ThreadGroup\n");
        return false;
    }
    gDvm.voffJavaLangThreadGroup_removeThread = meth->methodIndex;

    return true;
}
```

上述函数的实现代码保存在文件 Thread.c 中。

11. 初始化虚拟机使用的异常Java类库

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmExceptionStartup()初始化虚拟机使用的异常 Java 类库。此函数的具体实现代码如下所示：

```
bool dvmExceptionStartup(void)
{
    gDvm.classJavaLangThrowable =
        dvmFindSystemClassNoInit("Ljava/lang/Throwable;");
    gDvm.classJavaLangRuntimeException =
        dvmFindSystemClassNoInit("Ljava/lang/RuntimeException;");
    gDvm.classJavaLangError =
        dvmFindSystemClassNoInit("Ljava/lang/Error;");
    gDvm.classJavaLangStackTraceElement =
        dvmFindSystemClassNoInit("Ljava/lang/StackTraceElement;");
    gDvm.classJavaLangStackTraceElementArray =
        dvmFindArrayClass("[Ljava/lang/StackTraceElement;", NULL);
    if (gDvm.classJavaLangThrowable == NULL
      || gDvm.classJavaLangStackTraceElement == NULL
```

```
        || gDvm.classJavaLangStackTraceElementArray == NULL)
    {
        LOGE("Could not find one or more essential exception classes\n");
        return false;
    }

    /*
     * Find the constructor.  Note that, unlike other saved method lookups,
     * we're using a Method* instead of a vtable offset.  This is because
     * constructors don't have vtable offsets.  (Also, since we're creating
     * the object in question, it's impossible for anyone to sub-class it.)
     */
    Method *meth;
    meth = dvmFindDirectMethodByDescriptor(gDvm.classJavaLangStackTraceElement,
        "<init>", "(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;I)V");
    if (meth == NULL) {
        LOGE("Unable to find constructor for StackTraceElement\n");
        return false;
    }
    gDvm.methJavaLangStackTraceElement_init = meth;

    /* grab an offset for the stackData field */
    gDvm.offJavaLangThrowable_stackState =
        dvmFindFieldOffset(gDvm.classJavaLangThrowable,
            "stackState", "Ljava/lang/Object;");
    if (gDvm.offJavaLangThrowable_stackState < 0) {
        LOGE("Unable to find Throwable.stackState\n");
        return false;
    }

    /* and one for the message field, in case we want to show it */
    gDvm.offJavaLangThrowable_message =
        dvmFindFieldOffset(gDvm.classJavaLangThrowable,
            "detailMessage", "Ljava/lang/String;");
    if (gDvm.offJavaLangThrowable_message < 0) {
        LOGE("Unable to find Throwable.detailMessage\n");
        return false;
    }

    /* and one for the cause field, just 'cause */
    gDvm.offJavaLangThrowable_cause =
        dvmFindFieldOffset(gDvm.classJavaLangThrowable,
            "cause", "Ljava/lang/Throwable;");
    if (gDvm.offJavaLangThrowable_cause < 0) {
        LOGE("Unable to find Throwable.cause\n");
        return false;
    }
    return true;
}
```

上述函数的实现代码保存在文件 Exception.c 中。

12. 初始化虚拟机解释器使用的字符串哈希表

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmStringInternStartup()初始化虚拟机解释器使用的字符串哈希表。此函数的具体实现代码如下所示：

```
bool dvmStringInternStartup(void)
{
    gDvm.internedStrings = dvmHashTableCreate(256, NULL);
    if (gDvm.internedStrings == NULL)
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 Intern.c 中。

13. 初始化本地方法库的表

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmNativeStartup()来初始化本地方法库的表。此函数的具体实现代码如下所示：

```
bool dvmNativeStartup(void)
{
    gDvm.nativeLibs = dvmHashTableCreate(4, freeSharedLibEntry);
    if (gDvm.nativeLibs == NULL)
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 Native.c 中。

14. 初始化内部本地方法

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmInternalNativeStartup()初始化内部本地方法，建立哈希表，方便快速查找。此函数的具体实现代码如下所示：

```
bool dvmInternalNativeStartup()
{
    DalvikNativeClass *classPtr = gDvmNativeMethodSet;
    while (classPtr->classDescriptor != NULL) {
        classPtr->classDescriptorHash =
            dvmComputeUtf8Hash(classPtr->classDescriptor);
        classPtr++;
    }
    gDvm.userDexFiles = dvmHashTableCreate(2, dvmFreeDexOrJar);
    if (gDvm.userDexFiles == NULL)
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 native\InternalNative.cpp 中。

15. 初始化JNI调用表

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmJniStartup()初始化 JNI 调用表，以便快速找到本地方法调用的入口。此函数的具体实现代码如下所示：

```
bool dvmJniStartup(void)
{
#ifdef USE_INDIRECT_REF
    if (!dvmInitIndirectRefTable(&gDvm.jniGlobalRefTable,
            kGlobalRefsTableInitialSize, kGlobalRefsTableMaxSize,
            kIndirectKindGlobal))
        return false;
#else
    if (!dvmInitReferenceTable(&gDvm.jniGlobalRefTable,
            kGlobalRefsTableInitialSize, kGlobalRefsTableMaxSize))
        return false;
#endif
    dvmInitMutex(&gDvm.jniGlobalRefLock);
    gDvm.jniGlobalRefLoMark = 0;
    gDvm.jniGlobalRefHiMark = kGrefWaterInterval*2;
    if (!dvmInitReferenceTable(&gDvm.jniPinRefTable,
            kPinTableInitialSize, kPinTableMaxSize))
        return false;
    dvmInitMutex(&gDvm.jniPinRefLock);
    /*
     * Look up and cache pointers to some direct buffer classes, fields,
     * and methods.
     */
    Method *meth;
    ClassObject *platformAddressClass =
        dvmFindSystemClassNoInit("Lorg/apache/harmony/luni/platform/PlatformAddress;");
    ClassObject *platformAddressFactoryClass =
        dvmFindSystemClassNoInit(
          "Lorg/apache/harmony/luni/platform/PlatformAddressFactory;");
    ClassObject *directBufferClass =
        dvmFindSystemClassNoInit("Lorg/apache/harmony/nio/internal/DirectBuffer;");
    ClassObject *readWriteBufferClass =
        dvmFindSystemClassNoInit("Ljava/nio/ReadWriteDirectByteBuffer;");
    ClassObject *bufferClass =
        dvmFindSystemClassNoInit("Ljava/nio/Buffer;");
    if (platformAddressClass == NULL || platformAddressFactoryClass == NULL
      || directBufferClass == NULL || readWriteBufferClass == NULL
      || bufferClass == NULL)
    {
        LOGE("Unable to find internal direct buffer classes\n");
        return false;
    }
    gDvm.classJavaNioReadWriteDirectByteBuffer = readWriteBufferClass;
```

```
gDvm.classOrgApacheHarmonyNioInternalDirectBuffer = directBufferClass;
/* need a global reference for extended CheckJNI tests */
gDvm.jclassOrgApacheHarmonyNioInternalDirectBuffer =
    addGlobalReference((Object*) directBufferClass);
/*
 * We need a Method* here rather than a vtable offset, because
 * DirectBuffer is an interface class.
 */
meth = dvmFindVirtualMethodByDescriptor(
            gDvm.classOrgApacheHarmonyNioInternalDirectBuffer,
            "getEffectiveAddress",
            "()Lorg/apache/harmony/luni/platform/PlatformAddress;");
if (meth == NULL) {
    LOGE("Unable to find PlatformAddress.getEffectiveAddress\n");
    return false;
}
gDvm.methOrgApacheHarmonyNioInternalDirectBuffer_getEffectiveAddress = meth;
meth = dvmFindVirtualMethodByDescriptor(platformAddressClass,
            "toLong", "()J");
if (meth == NULL) {
    LOGE("Unable to find PlatformAddress.toLong\n");
    return false;
}
gDvm.voffOrgApacheHarmonyLuniPlatformPlatformAddress_toLong =
    meth->methodIndex;
meth = dvmFindDirectMethodByDescriptor(platformAddressFactoryClass,
            "on",
            "(I)Lorg/apache/harmony/luni/platform/PlatformAddress;");
if (meth == NULL) {
    LOGE("Unable to find PlatformAddressFactory.on\n");
    return false;
}
gDvm.methOrgApacheHarmonyLuniPlatformPlatformAddress_on = meth;
meth = dvmFindDirectMethodByDescriptor(readWriteBufferClass,
            "<init>",
            "(Lorg/apache/harmony/luni/platform/PlatformAddress;II)V");
if (meth == NULL) {
    LOGE("Unable to find ReadWriteDirectByteBuffer.<init>\n");
    return false;
}
gDvm.methJavaNioReadWriteDirectByteBuffer_init = meth;
gDvm.offOrgApacheHarmonyLuniPlatformPlatformAddress_osaddr =
    dvmFindFieldOffset(platformAddressClass, "osaddr", "I");
if (gDvm.offOrgApacheHarmonyLuniPlatformPlatformAddress_osaddr < 0) {
    LOGE("Unable to find PlatformAddress.osaddr\n");
    return false;
}
gDvm.offJavaNioBuffer_capacity =
    dvmFindFieldOffset(bufferClass, "capacity", "I");
```

```
    if (gDvm.offJavaNioBuffer_capacity < 0) {
        LOGE("Unable to find Buffer.capacity\n");
        return false;
    }
    gDvm.offJavaNioBuffer_effectiveDirectAddress =
        dvmFindFieldOffset(bufferClass, "effectiveDirectAddress", "I");
    if (gDvm.offJavaNioBuffer_effectiveDirectAddress < 0) {
        LOGE("Unable to find Buffer.effectiveDirectAddress\n");
        return false;
    }
    return true;
}
```

上述函数的实现代码保存在文件 Jni.c 中。

16. 缓存Java类库里的反射类

在 Dalvik 虚拟机的初始化过程中，使用函数 dvmReflectStartup()缓存 Java 类库里的反射类。此函数的具体实现代码如下所示：

```
bool dvmReflectStartup(void)
{
    gDvm.classJavaLangReflectAccessibleObject =
        dvmFindSystemClassNoInit("Ljava/lang/reflect/AccessibleObject;");
    gDvm.classJavaLangReflectConstructor =
        dvmFindSystemClassNoInit("Ljava/lang/reflect/Constructor;");
    gDvm.classJavaLangReflectConstructorArray =
        dvmFindArrayClass("[Ljava/lang/reflect/Constructor;", NULL);
    gDvm.classJavaLangReflectField =
        dvmFindSystemClassNoInit("Ljava/lang/reflect/Field;");
    gDvm.classJavaLangReflectFieldArray =
        dvmFindArrayClass("[Ljava/lang/reflect/Field;", NULL);
    gDvm.classJavaLangReflectMethod =
        dvmFindSystemClassNoInit("Ljava/lang/reflect/Method;");
    gDvm.classJavaLangReflectMethodArray =
        dvmFindArrayClass("[Ljava/lang/reflect/Method;", NULL);
    gDvm.classJavaLangReflectProxy =
        dvmFindSystemClassNoInit("Ljava/lang/reflect/Proxy;");
    if (gDvm.classJavaLangReflectAccessibleObject == NULL
      || gDvm.classJavaLangReflectConstructor == NULL
      || gDvm.classJavaLangReflectConstructorArray == NULL
      || gDvm.classJavaLangReflectField == NULL
      || gDvm.classJavaLangReflectFieldArray == NULL
      || gDvm.classJavaLangReflectMethod == NULL
      || gDvm.classJavaLangReflectMethodArray == NULL
      || gDvm.classJavaLangReflectProxy == NULL)
    {
        LOGE("Could not find one or more reflection classes\n");
        return false;
    }
```

```
gDvm.methJavaLangReflectConstructor_init =
    dvmFindDirectMethodByDescriptor(gDvm.classJavaLangReflectConstructor, "<init>",
    "(Ljava/lang/Class;[Ljava/lang/Class;[Ljava/lang/Class;I)V");
gDvm.methJavaLangReflectField_init =
    dvmFindDirectMethodByDescriptor(gDvm.classJavaLangReflectField, "<init>",
    "(Ljava/lang/Class;Ljava/lang/Class;Ljava/lang/String;I)V");
gDvm.methJavaLangReflectMethod_init =
    dvmFindDirectMethodByDescriptor(gDvm.classJavaLangReflectMethod, "<init>",
    "(Ljava/lang/Class;[Ljava/lang/Class;[Ljava/lang/Class;
      Ljava/lang/ Class;Ljava/lang/String;I)V");
if (gDvm.methJavaLangReflectConstructor_init == NULL
  || gDvm.methJavaLangReflectField_init == NULL
  || gDvm.methJavaLangReflectMethod_init == NULL)
{
    LOGE("Could not find reflection constructors\n");
    return false;
}

gDvm.classJavaLangClassArray =
    dvmFindArrayClass("[Ljava/lang/Class;", NULL);
gDvm.classJavaLangObjectArray =
    dvmFindArrayClass("[Ljava/lang/Object;", NULL);
if (gDvm.classJavaLangClassArray == NULL
  || gDvm.classJavaLangObjectArray == NULL)
{
    LOGE("Could not find class-array or object-array class\n");
    return false;
}

gDvm.offJavaLangReflectAccessibleObject_flag =
    dvmFindFieldOffset(gDvm.classJavaLangReflectAccessibleObject, "flag",
        "Z");

gDvm.offJavaLangReflectConstructor_slot =
    dvmFindFieldOffset(gDvm.classJavaLangReflectConstructor, "slot", "I");
gDvm.offJavaLangReflectConstructor_declClass =
    dvmFindFieldOffset(gDvm.classJavaLangReflectConstructor,
        "declaringClass", "Ljava/lang/Class;");

gDvm.offJavaLangReflectField_slot =
    dvmFindFieldOffset(gDvm.classJavaLangReflectField, "slot", "I");
gDvm.offJavaLangReflectField_declClass =
    dvmFindFieldOffset(gDvm.classJavaLangReflectField,
        "declaringClass", "Ljava/lang/Class;");

gDvm.offJavaLangReflectMethod_slot =
    dvmFindFieldOffset(gDvm.classJavaLangReflectMethod, "slot", "I");
gDvm.offJavaLangReflectMethod_declClass =
    dvmFindFieldOffset(gDvm.classJavaLangReflectMethod,
```

```
        "declaringClass", "Ljava/lang/Class;");

    if (gDvm.offJavaLangReflectAccessibleObject_flag < 0
      || gDvm.offJavaLangReflectConstructor_slot < 0
      || gDvm.offJavaLangReflectConstructor_declClass < 0
      || gDvm.offJavaLangReflectField_slot < 0
      || gDvm.offJavaLangReflectField_declClass < 0
      || gDvm.offJavaLangReflectMethod_slot < 0
      || gDvm.offJavaLangReflectMethod_declClass < 0)
    {
        LOGE("Could not find reflection fields\n");
        return false;
    }
    if (!dvmReflectProxyStartup())
        return false;
    if (!dvmReflectAnnotationStartup())
        return false;
    return true;
}
```

上述函数的实现代码保存在文件 reflect\Reflect.c 中。

17. 剩余类的初始化工作

经过前面的初始化函数处理之后，接着把下面的类先进行初始化操作：

```
static const char *earlyClasses[] = {
    "Ljava/lang/InternalError;",
    "Ljava/lang/StackOverflowError;",
    "Ljava/lang/UnsatisfiedLinkError;",
    "Ljava/lang/NoClassDefFoundError;",
    NULL
};
```

调用 dvmFindSystemClassNoInit 函数来初始化这些类。
接着调用函数 dvmValidateBoxClasses()来初始化下列 Java 基本类型库：

```
static const char *classes[] = {
    "Ljava/lang/Boolean;",
    "Ljava/lang/Character;",
    "Ljava/lang/Float;",
    "Ljava/lang/Double;",
    "Ljava/lang/Byte;",
    "Ljava/lang/Short;",
    "Ljava/lang/Integer;",
    "Ljava/lang/Long;",
    NULL
};
```

这些类调用函数，不是使用系统函数来初始化，而是调用函数 dvmFindClassNoInit()来初

始化。此函数的实现代码如下所示：

```
ClassObject* dvmFindClassNoInit(const char *descriptor, Object *loader)
{
    assert(descriptor != NULL);
    //assert(loader != NULL);

    LOGVV("FindClassNoInit '%s' %p\n", descriptor, loader);

    if (*descriptor == '[') {
        /*
         * Array class.  Find in table, generate if not found.
         */
        return dvmFindArrayClass(descriptor, loader);
    } else {
        /*
         * Regular class. Find in table, load if not found.
         */
        if (loader != NULL) {
            return findClassFromLoaderNoInit(descriptor, loader);
        } else {
            return dvmFindSystemClassNoInit(descriptor);
        }
    }
}
```

调用函数 dvmPrepMainForJni()准备主线程里的解释栈时可以调用 JNI 的方法，此函数的实现代码如下所示：

```
bool dvmPrepMainForJni(JNIEnv *pEnv)
{
    Thread *self;

    /* main thread is always first in list at this point */
    self = gDvm.threadList;
    assert(self->threadId == kMainThreadId);

    /* create a "fake" JNI frame at the top of the main thread interp stack */
    if (!createFakeEntryFrame(self))
        return false;

    /* fill these in, since they weren't ready at dvmCreateJNIEnv time */
    dvmSetJniEnvThreadId(pEnv, self);
    dvmSetThreadJNIEnv(self, (JNIEnv*)pEnv);

    return true;
}
```

调用函数 registerSystemNatives()来注册 Java 库里的 JNI 方法，此函数的实现代码如下所示：

```
static bool registerSystemNatives(JNIEnv *pEnv)
{
    Thread *self;

    /* main thread is always first in list */
    self = gDvm.threadList;

    /* must set this before allowing JNI-based method registration */
    self->status = THREAD_NATIVE;

    if (jniRegisterSystemMethods(pEnv) < 0) {
        LOGE("jniRegisterSystemMethods failed");
        return false;
    }

    /* back to run mode */
    self->status = THREAD_RUNNING;

    return true;
}
```

调用函数 dvmCreateStockExceptions()分配异常出错的内存，此函数的实现代码如下所示：

```
bool dvmCreateStockExceptions(void)
{
    /*
     * Pre-allocate some throwables.  These need to be explicitly added
     * to the GC's root set (see dvmHeapMarkRootSet()).
     */
    gDvm.outOfMemoryObj = createStockException("Ljava/lang/OutOfMemoryError;",
        "[memory exhausted]");
    dvmReleaseTrackedAlloc(gDvm.outOfMemoryObj, NULL);
    gDvm.internalErrorObj = createStockException("Ljava/lang/InternalError;",
        "[pre-allocated]");
    dvmReleaseTrackedAlloc(gDvm.internalErrorObj, NULL);
    gDvm.noClassDefFoundErrorObj =
        createStockException("Ljava/lang/NoClassDefFoundError;", NULL);
    dvmReleaseTrackedAlloc(gDvm.noClassDefFoundErrorObj, NULL);

    if (gDvm.outOfMemoryObj == NULL || gDvm.internalErrorObj == NULL
      || gDvm.noClassDefFoundErrorObj == NULL)
    {
        LOGW("Unable to create stock exceptions\n");
        return false;
    }

    return true;
}
```

调用函数 dvmPrepMainThread()实现解释器主线程的初始化，此函数的实现代码如下所示：

```
bool dvmPrepMainThread(void)
{
    Thread *thread;
    Object *groupObj;
    Object *threadObj;
    Object *vmThreadObj;
    StringObject *threadNameStr;
    Method *init;
    JValue unused;
    LOGV("+++ finishing prep on main VM thread\n");
    /* main thread is always first in list at this point */
    thread = gDvm.threadList;
    assert(thread->threadId == kMainThreadId);
    /*
     * Make sure the classes are initialized.  We have to do this before
     * we create an instance of them.
     */
    if (!dvmInitClass(gDvm.classJavaLangClass)) {
        LOGE("'Class' class failed to initialize\n");
        return false;
    }
    if (!dvmInitClass(gDvm.classJavaLangThreadGroup)
      || !dvmInitClass(gDvm.classJavaLangThread)
      || !dvmInitClass(gDvm.classJavaLangVMThread))
    {
        LOGE("thread classes failed to initialize\n");
        return false;
    }
    groupObj = dvmGetMainThreadGroup();
    if (groupObj == NULL)
        return false;
    /*
     * Allocate and construct a Thread with the internal-creation
     * constructor.
     */
    threadObj = dvmAllocObject(gDvm.classJavaLangThread, ALLOC_DEFAULT);
    if (threadObj == NULL) {
        LOGE("unable to allocate main thread object\n");
        return false;
    }
    dvmReleaseTrackedAlloc(threadObj, NULL);

    threadNameStr = dvmCreateStringFromCstr("main", ALLOC_DEFAULT);
    if (threadNameStr == NULL)
        return false;
    dvmReleaseTrackedAlloc((Object*)threadNameStr, NULL);
    init = dvmFindDirectMethodByDescriptor(gDvm.classJavaLangThread, "<init>",
            "(Ljava/lang/ThreadGroup;Ljava/lang/String;IZ)V");
```

```
    assert(init != NULL);
    dvmCallMethod(thread, init, threadObj, &unused, groupObj, threadNameStr,
        THREAD_NORM_PRIORITY, false);
    if (dvmCheckException(thread)) {
        LOGE("exception thrown while constructing main thread object\n");
        return false;
    }
```

调用函数 dvmDebuggerStartup()进行调试器的初始化，此函数的实现代码如下所示：

```
bool dvmDebuggerStartup(void)
{
    gDvm.dbgRegistry = dvmHashTableCreate(1000, NULL);
    return (gDvm.dbgRegistry != NULL);
}
```

调用 dvmInitZygote()或者 dvmInitAfterZygote()来初始化线程的模式，此函数的实现代码如下所示：

```
static bool dvmInitZygote(void)
{
    /* zygote goes into its own process group */
    setpgid(0,0);
    return true;
}
bool dvmInitAfterZygote(void)
{
    u8 startHeap, startQuit, startJdwp;
    u8 endHeap, endQuit, endJdwp;
    startHeap = dvmGetRelativeTimeUsec();
    /*
     * Post-zygote heap initialization, including starting
     * the HeapWorker thread.
     */
    if (!dvmGcStartupAfterZygote())
        return false;
    endHeap = dvmGetRelativeTimeUsec();
    startQuit = dvmGetRelativeTimeUsec();
    /* start signal catcher thread that dumps stacks on SIGQUIT */
    if (!gDvm.reduceSignals && !gDvm.noQuitHandler) {
        if (!dvmSignalCatcherStartup())
            return false;
    }
    /* start stdout/stderr copier, if requested */
    if (gDvm.logStdio) {
        if (!dvmStdioConverterStartup())
            return false;
    }
    endQuit = dvmGetRelativeTimeUsec();
    startJdwp = dvmGetRelativeTimeUsec();
```

```
    /*
     * Start JDWP thread.  If the command-line debugger flags specified
     * "suspend=y", this will pause the VM.  We probably want this to
     * come last.
     */
    if (!dvmInitJDWP()) {
        LOGD("JDWP init failed; continuing anyway\n");
    }
    endJdwp = dvmGetRelativeTimeUsec();
    LOGV("thread-start heap=%d quit=%d jdwp=%d total=%d usec\n",
        (int)(endHeap-startHeap), (int)(endQuit-startQuit),
        (int)(endJdwp-startJdwp), (int)(endJdwp-startHeap));
#ifdef WITH_JIT
    if (gDvm.executionMode == kExecutionModeJit) {
        if (!dvmCompilerStartup())
            return false;
    }
#endif
    return true;
}
```

调用函数 dvmCheckException()检查是否有异常情况出现，函数的具体实现代码略。

5.2.3 启动Zygote

在本书前面的内容中，已经分析过启动孕育进程 Zygote 的基本源码。在本节的内容中，将详细分析 Zygote 进程的内部启动过程。

(1) 在 init.rc 中配置 zygote 启动参数

init.rc 保存在设备的根目录下，我们可以使用 adb pull /init.rc ~/Desktop 命令取出该文件，文件中与 Zygote 相关的配置信息如下所示：

```
service zygote /system/bin/app process -Xzygote
/system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

(2) 启动 Socket 服务端口

当 Zygote 服务从 app_process 启动后，会启动一个 Dalvik 虚拟机。因为虚拟机执行的第一个 Java 类是 ZygoteInit.java，所以接下来的过程就从类 ZygoteInit 中的函数 main()开始讲起。

函数 main()中做的第一个重要工作就是启动一个 Socket 服务端口，该 Socket 端口用于接收启动新进程的命令：

- 在静态函数 registerZygoteSocket()中，完成启动 Socket 服务端口的功能。
- 当准备好 LocalServerSocket 端口后，在函数 main()中调用 runSelectLoopMode()进入非阻塞读操作，该函数会先将 ServerSocket 加入到被监测的文件描述符列表中，然后在

while(true)循环中将该文件描述符添加到 select 的列表中，并调用 ZygoteConnection 类的 runOnce()函数处理每一个 Socket 接收到的命令。

- 在 SystemServer 进程中创建一个 Socket 客户端，具体的代码是在文件 Process.java 中实现，而调用 Process 类的工作是在类 AmS 中的 startProcessLocked()函数中实现的。
- 函数 start()内部又调用了静态函数 startViaZygote()，该函数的实体是使用一个本地 Socket 向 Zygote 中的 Socket 发送进行启动的命令，其执行流程如下所示。

① 将 startViaZygote()的函数参数转换为一个 ArrayList<String>列表。

② 然后再构造出一个 LocalSocket 本地 Socket 接口。

③ 通过该 LocalSocket 对象构造出一个 BufferedWriter 对象。

④ 通过该对象将 ArralyList<String>中的参数传递给 Zygote 的 LocalServerSocket。

⑤ 在 Zygote 端调用 Zygote.forkAndSpecialize()函数，孕育出一个新的应用进程。

(3)　加载 preload-classes

在类 ZygoteInit 的函数 main()中，创建完 Socket 服务端后还不能马上孕育新的进程，因为这个“卵”中还没有预装 Framework 大部分的类及资源。

预装的类列表是在 framework.jar 中的一个文本文件列表，名称为 preload-classes，该列表的原始定义在文本文件 frameworks/base/preload-classes 中，而该文件又是通过如下类生成的：

```
frameworks/base/tools/preload/WritePreloadedClassFile.java
```

生成 preload-classes 的方法是在 Android 根目录下执行如下命令：

```
$java -Xss512M -cp /path/to/preload.jar WritePreloadedClassFile /path/to/.compiled
1517 classes were loaded by more than one app.
Added 147 more to speed up applications.
1664 total classes will be preloaded.
Writing object model...
Done!
```

在上述命令中，/path/to/preload.jar 是指如下文件：

```
out/host/darwin-x86/framework/preload.jar
```

上述.jar 文件是由 frameworks/base/tools/preload 子项目编译而成的。

/path/to/.compiled/是指如下目录下的几个.compiled 文件：

```
frameworks/base/tools/preload
```

① 参数“-Xss”：用于执行该程序所需要的 Java 虚拟机栈大小，此处使用 512MB，默认的大小不能满足该程序的运行，会抛出 java.lang.StackOverflowError 错误信息。

② WritePreloadedClassFile：表示要执行的具体类。

当执行完以上命令后，会在 frameworks/base 目录下产生 preload-classes 文本文件。从该命令的执行情况来看，预装的 Java 类信息包含在.compiled 文件中，而这个文件却是一个二进制文件，尽管我们目前能够确知如何产生 preload-classes，但却无法明确这个.compiled 文件是如何产生的。

在 Android 项目组内部可能会存在一个测试项目，一旦运行该项目，就会装载一些 Java 类。当然，这些 Java 类是测试项目中的程序代码主动装载的，而这些程序代码被认为是大多数

Android 程序运行时都会执行的代码。一旦该运行环境建立后，Dalvik 虚拟机内存中就记录了所有被装载的 Java 类，然后该测试项目会使用一个特别的工具从虚拟机内存中读取所有装载过的类信息，并生成被编译的文件。当然，这只是一种假设。

在编译 Android 源码的时候，会最终把 preload-classes 文件打包到 framework.jar 中。这样，有了这个列表后，ZygoteInit 中通过调用 preloadClasses()完成装载这些类。装载的方法很简单，就是读取 preload-classes 列表中的每一行，因为每一行代表了一个具体的类，然后调用 Class.forName()装载目标类。在装载的过程中，忽略以#开始的目标类，并忽略换行符及空格。

(4) 加载 preload-resources

preload-resources 是在如下文件中被定义的：

```
frameworks/base/core/res/res/values/arrays.xml
```

在 preload-resources 中包含了两类资源，一类是 drawable 资源，另一类是 color 资源，下面是对应的代码：

```
<array name="preloaded drawables">
    <item>@drawable/sym_def_app_icon</item>
    ...
</array>
<array name="preloaded_color_state_lists">
    <item>@color/hint_foreground_dark</item>
    ...
</array>
```

加载这些资源的功能是在函数 preloadResources()中实现的，在该函数中分别调用了如下两个函数来加载这两类资源：preloadDrawables()和 preloadColorStateLists()。

具体的加载原理非常简单，就是把这些资源读出来，放到一个全局变量中，只要该类对象不被销毁，这些全局变量就会一直保存。

通过全局变量 mResources 来保存 Drawable 资源，该变量的类型是 Resources 类，由于在该类内部会保存一个 Drawable 资源列表，因此实际上是在 Resources 内部缓存这些 Drawable 资源的。保存 Color 资源的全局变量的功能也是 mResources 实现的。同样，在类 Resources 内部也有一个 Color 资源列表。

(5) 使用 fork 启动新进程

fork 是 Linux 系统中的一个系统调用，其功能是复制当前进程并产生一个新的进程。除了进程 id 不同，新的进程将拥有与原始进程完全相同的进程信息。进程信息包括该进程所打开的文件描述符列表、所分配的内存等。当创建新进程后，两个进程将共享已经分配的内存空间，直到其中一个需要向内存中写入数据时，操作系统才负责复制一份目标地址空间，并将要写的数据写入到新的地址中，这就是“copy-on-write(仅当写的时候才复制)”机制，这种机制可以最大限度地在多个进程中共享物理内存。

在所有的操作系统中，都存在一个程序装载器，程序装载器一般会作为操作系统的一部分，并由 Shell 程序调用。当内核启动后，会首先启动 Shell 程序。

常见的 Shell 程序包含如下两大类：

- 命令行界面的。
- 窗口界面的。

Windows 系统中的 Shell 程序就是桌面程序，Ubuntu 系统中的 Shell 程序就是 GNOME 桌面程序。当启动 Shell 程序后，用户可以双击桌面图标启动指定的应用程序，而在操作系统内部，启动新的进程包含如下三个过程。

① 第一个过程，内核创建一个进程数据结构，用于表示将要启动的进程。

② 第二个过程，内核调用程序装载器函数，从指定的程序文件读取程序代码，并将这些程序代码装载到预先设定的内存地址。

③ 第三个过程，装载完毕后，内核将程序指针指向到目标程序地址的入口处开始执行指定的进程。当然，实际的过程会考虑更多的细节，不过大致思路就是这么简单。

在一般情况下，没有必要复制进程，而是按照以上 3 个过程创建新进程，但当满足条件时，则由于函数 fork()是 Linux 的系统调用，Android 中的 Java 层仅仅是对该调用进行了 JNI 封装而已，因此，接下来以一段 C 代码来介绍使用函数 fork()的过程，以便读者对该函数有更具体的认识：

```
/**
*FileName: abc.c
*/
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t pid;
    printf("pid = %d, Take camera, by subway, take air! \n", getpid());
    pid = fork();
    if(pid > 0) {
        printf("pid=%d, 我是无敌! \n", getpid());
        pid = fork();
        if(!pid) printf("pid=%d, 去看 AA! \n", getpid());
    }
    else if (!pid) printf("pid=%d, 去看 BB! \n", getpid());
    else if (pid == -1) perror("fork");
    getchar();
}
```

执行上述代码后会输出：

```
$ ./abc.bin
pid = 3927, Take camera, by subway, take air!
pid=3927, 我是无敌!
pid=3929, 去看 AA!
pid=3930, 去看 BB!
```

函数 fork()的返回值与普通函数调用完全不同，具体说明如下所示：

- 当返回值大于 0 时，代表的是父进程。
- 当等于 0 时，代表的是被复制的进程。

也就是说，父进程和子进程的代码都在该 C 文件中，只是不同的进程执行不同的代码，而进程是靠 fork()的返回值进行区分的。

由以上执行结果可以看出，第一次调用 fork()时复制了一个“看 AA”进程，然后在父进

程中再次调用 fork()复制了“看 BB”的进程，三者都有各自不同的进程 id。

Zygote 进程是上述演示代码中的“精灵进程”。在文件 ZygoteInit.java 中复制新进程是通过在函数 runSelectLoopMode()中调用类 ZygoteConnection 的函数 runOnce()完成的，在该函数中通过调用 forkAndSpecialize()函数来复制一个新的进程。

函数 forkAndSpecialize()是一个 native(本地)函数，其内部的执行原理与上面的 C 代码类似。当新进程被创建好后，还需要做一些“完善”工作。因为当 Zygote 复制新进程时，已经创建了一个 Socket 服务端，而这个服务端是不应该被新进程使用的，否则系统中会有多个进程接收 Socket 客户端的命令。因此，新进程被创建好后，首先需要在新进程中关闭该 Socket 服务端，并调用新进程中指定的 Class 文件的 main()函数作为新进程的入口点。而这些正是在调用函数 forkAndSpecialize()后根据返回值 pid 完成的。当 pid 等于 0 时，代表的是子进程，函数 handleChildProc()会从指定 Class 文件的 main()函数处开始执行。新的进程会完全脱离 Zygote 进程的孕育过程，成为一个真正的应用进程。

5.2.4 启动SystemServer进程

SystemServer 进程是 Zygote 孕育出的第一个进程，该进程是从 ZygoteInit.java 的 main()函数中调用 startSystemServer()开始的。与启动普通进程的差别在于，zygote 类为启动 SystemServer 提供了专门的函数 startSystemServer()，而不是使用标准的 forkAndSpecilize()函数。同时，启动 SystemServer 进程后，首先要做的事情与普通进程也有所差别。

函数 startSystemServer()的主要功能如下所示。

① 定义了一个 String[]数组，数组中包含了要启动的进程的相关信息，其中最后一项指定新进程启动后装载的第一个 Java 类，此处即为类 com.android.server.SystemServer。

② 调用 forkSystemServer()从当前的 Zygote 进程孕育出新的进程。该函数是一个 native 函数，其作用与 forkAndSpecilize()相似。

③ 启动新进程后，在函数 handleSystemServerProcess()中主要完成如下两件事情：

- 关闭 Socket 服务端。
- 执行 com.android.server.SystemServer 类中的 main()函数。

除了这两个主要事情外，还做了一些额外的运行环境配置，这些配置主要在函数 commonInit()和函数 zygoteInitNative()中完成。一旦配置好 SystemServer 的进程环境后，就从类 SystemServer 中的 main()函数开始运行。

(1) 启动各种系统服务线程

SystemServer 进程在 Android 运行环境中扮演了“中枢”的角色，在 APK 应用中能够直接交互的大部分系统服务都在这个进程中运行，例如 WindowManagerServer(Wms)、ActivityManagerSystemService(AmS)、PackageManagerServer(PmS)等常见的应用，这些系统服务都是以一个线程的方式存在于 SystemServer 进程中的。下面就来介绍到底都有哪些服务线程，及其启动的顺序。

SystemServer 中的 main()函数首先调用的是函数 init1()，这是一个 native 函数，内部会进行一些与 Dalvik 虚拟机相关的初始化工作。该函数执行完毕后，其内部会调用 Java 端的 init2()函数，这就是为什么 Java 源码中没有引用 init2()的地方，主要的系统服务都是在 init2()函数中完成的。

该函数首先创建了一个 ServerThread 对象，该对象是一个线程，然后直接运行该线程，从 ServerThread 的 run()方法内部开始真正启动各种服务线程。基本上，每个服务都有对应的 Java 类，从编码规范的角度来看，启动这些服务的模式可归类为如下三种。

- 模式一：是指直接使用构造函数构造一个服务，由于大多数服务都对应一个线程，因此，在构造函数内部就会创建一个线程并自动运行。
- 模式二：是指服务类会提供一个 getInstance()方法，通过该方法获取该服务对象，这样的好处是保证系统中仅包含一个该服务对象。
- 模式三：是指从服务类的 main()函数中开始执行。

无论以上何种模式，当创建了服务对象后，有时可能还需要调用该服务类的 init()函数或者 systemReady()函数来完成该对象的启动，当然，这些都是服务类内部自定义的。为了区分以上启动的不同，以下采用一种新的方式来描述该启动过程。

在表 5-2 中列出了 SystemServer 中启动的所有服务，以及这些服务的启动模式。

表 5-2　SystemServer中启动的服务

服务类名称	作用描述	启动模式
EntropyService	提供伪随机数	1.0
PowerManagerService	电源管理服务	1.2/3
ActivityManagerService	最核心的服务之一，管理 Activity	自定义
TelephonyRegistry	注册电话模块的事件响应，比如重启、关闭、启动等	1.0
PackageManagerService	程序包管理服务	3.3
AccountManagerService	账户管理服务，是指联系人账户，而不是 Linux 系统的账户	1.0
ContentService	ContentProvider 服务，提供跨进程数据交换	3.0
BatteryService	电池管理服务	1.0
LightsService	自然光强度感应传感器服务	1.0
VibratorService	震动器服务	1.0
AlarmManagerService	定时器管理服务，提供定时提醒服务	1.0
WindowManagerService	Framework 最核心的服务之一，负责窗口管理	3.3
BluetoothService	蓝牙服务	1.0+
DevicePolicyManagerService	提供一些系统级别的设置及属性	1.3
StatusBarManagerService	状态栏管理服务	1.3
ClipboardService	系统剪贴板服务	1.0
InputMethodManagerService	输入法管理服务	1.0
NetStatService	网络状态服务	1.0
NetworkManagementService	网络管理服务	NMS.create()
ConnectivityService	网络连接管理服务	2.3
ThrottleService	暂不清楚其作用	1.3
AccessibilityManagerService	辅助管理程序截获所有的用户输入，并根据这些输入给用户一些额外的反馈，起到辅助的效果	1.0

续表

服务类名称	作用描述	启动模式
MountService	挂载服务，可通过该服务调用 Linux 层面的 mount 程序	1.0
NotificationManagerService	通知栏管理服务，Android 中的通知栏和状态栏在一起，只是界面上前者在左边，后者在右边	1.3
DeviceStorageMonitorService	磁盘空间状态检测服务	1.0
LocationManagerService	地理位置服务	1.3
SearchManagerService	搜索管理服务	1.0
DropBoxManagerService	通过该服务访问 Linux 层面的 Dropbox 程序	1.0
WallpaperManagerService	墙纸管理服务，墙纸不等同于桌面背景，在 View 系统内部，墙纸可以作为任何窗口的背景	1.3
AudioService	音频管理服务	1.0
BackupManagerService	系统备份服务	1.0
AppWidgetService	Widget 服务	1.3
RecognitionManagerService	身份识别服务	1.3
DiskStatsService	磁盘统计服务	1.0

AmS 的启动模式如下所示：

- 调用函数 main()返回一个 Context 对象，而不是 AmS 服务本身。
- 调用 AmS.setSystemProcess()。
- 调用 AmS.installProviders()。
- 调用 systemReady()，当 AmS 执行完 systemReady()后，会相继启动相关联服务的 systemReady()函数，完成整体初始化。

(2) 启动第一个 Activity

当启动以上服务线程后，ActivityManagerService(AmS)服务是以 systemReady()调用完成最后启动的，而在 AmS 的函数 systemReady()内部的最后一段代码则发出了启动任务队列中最上面一个 Activity 的消息。因为在系统刚启动时，mMainStack 队列中并没有任何 Activity 对象，所以在类 ActivityStack 中将调用函数 startHomeActivityLocked()。

在开机后，系统从哪个 Activity 开始执行这一动作，完全取决于 mMainStack 中的第一个 Activity 对象。如果在 ActivityManagerService 启动时能够构造一个 Activity 对象(并不是说构造出一个 Activity 类的对象)，并将其放到 mMainStack 中，那么第一个运行的 Activity 就是这个 Activity，这一点不像其他操作系统中通过设置一个固定程序作为第一个启动程序。

在 AmS 的 startHomeActivityLocked()中，系统发出了一个 category 字段包含 CATEGORY_HOME 的 intent。

无论是哪个应用程序，只要声明自己能够响应该 intent，那么就可以被认为是 Home 程序，这就是为什么在 Android 领域中会存在各种“Home 程序”的原因。系统并没有给任何程序赋予“Home”特权，而只是把这个权利交给了用户。当在系统中有多个程序能够响应该 intent 时，系统会弹出一个对话框，请求用户选择启动哪个程序，并允许用户记住该选择，从而使得以后每次按 Home 键后，都启动相同的 Activity。这就是第一个 Activity 的启动过程。

5.2.5　加载class类文件

Java 的源代码经过编译后，会生成“.class”格式的文件，即字节码文件。然后在 Android 中使用 dx 工具将其转换为后缀为“.jar”格式的 Dex 类型文件。Dalvik 虚拟机负责解释并执行编译后的字节码。在解释执行字节码之前，当然要读取文件，分析文件的内容，得到字节码，然后才能解释和执行。在整个的加载过程中，最为重要的就是对 Class 的加载，Class 包含 Method，Method 又包含 code。通过对 Class 的加载，我们即可获得所需执行的字节码。在本节的内容中，将从 Dexfile 文件分析及 Class 加载中的数据结构入手，结合主要流程，对整个加载过程进行分析。

(1)　DexFile 在内存中的映射

在 Android 系统中，Java 源文件会被编译为“.jar”格式的 Dex 类型文件，在代码中称为 Dexfile。在加载 Class 之前，必先读取相应的 JAR 文件。通常我们使用 read()函数来读取文件中的内容，但在 Dalvik 中使用 mmap()函数。与 read()不同的是，mmap()函数会将 Dex 文件映射到内存中，这样，通过普通的内存读取操作，即可访问 Dexfile 中的内容。

Dexfile 的文件格式如图 5-5 所示，主要由三部分组成：头部、索引、数据。通过头部可知索引的位置和数目，可知数据区的起始位置。其中 classDefsOff 指定了 ClassDef 在文件中的起始位置，dataOff 指定了数据在文件中的起始位置，ClassDef 可理解为 Class 的索引。通过读取 ClassDef 可获知 Class 的基本信息，其中 classDataOff 指定了 Class 数据在数据区的位置。

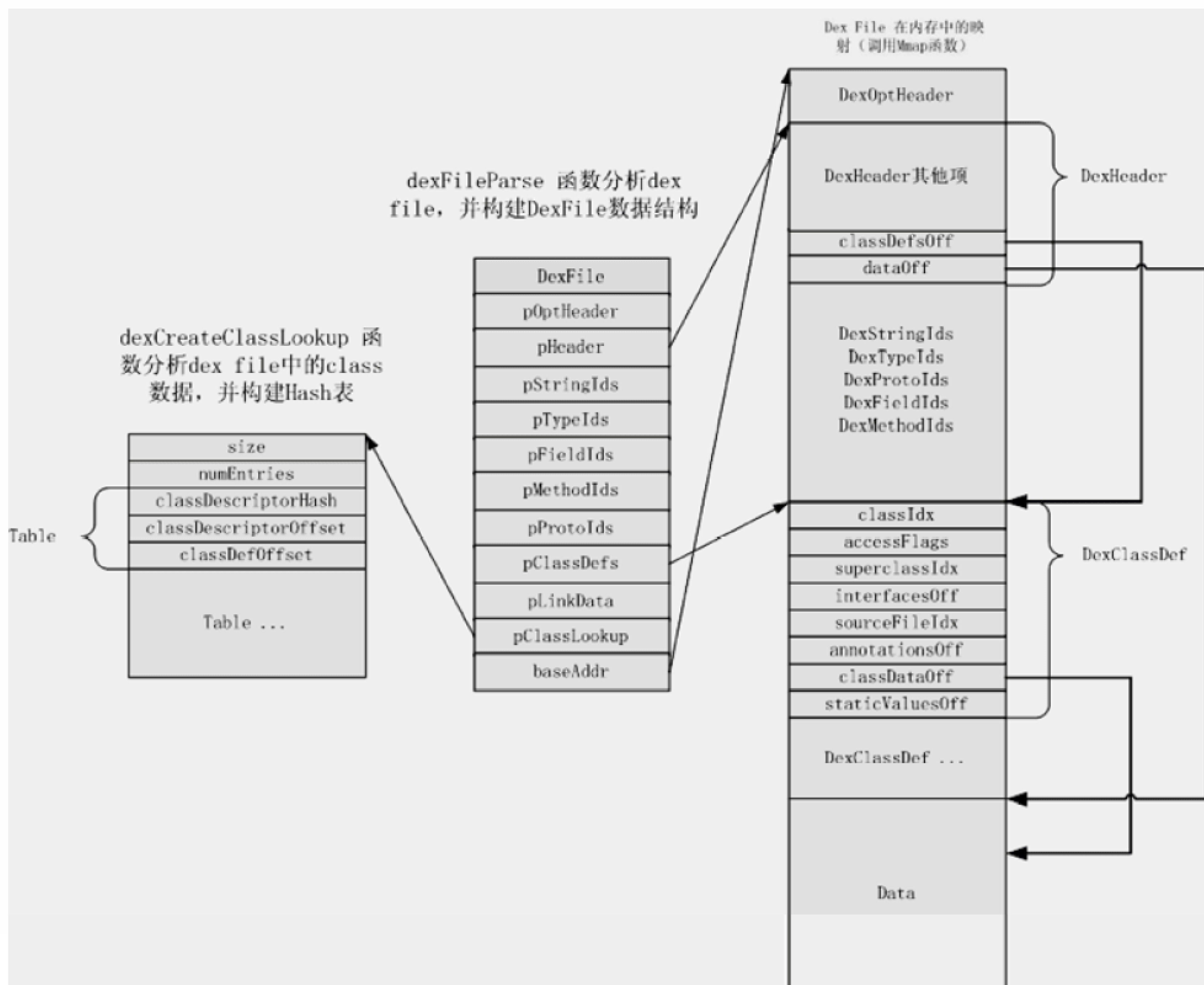


图 5-5　Dexfile的文件格式

在将 Dexfile 文件映射到内存后，会调用 dexFileParse()函数对其分析，分析的结果存放于名为 DexFile 的数据结构中。DexFile 中的 baseAddr 指向映射区的起始位置，pClassDefs 指向 ClassDefs(即 class 索引)的起始位置。由于在查找 class 时，都是使用 class 的名字进行查找的，所以为了加快查找速度，创建了一个 hash 表。在 hash 表中对 class 名字进行 hash，并生成 index。这些操作都是在对文件解析时完成的，这样虽然在加载过程中比较耗时，但是在运行过程中却可节省大量的查找时间。

解析完毕后，接下来开始加载 class 文件。在此需要将加载类用 ClassObject 来保存，所以需要先分析与 ClassObject 相关的几个数据结构。

首先在文件 Object.h 中可以看到如下对结构体 Object 的定义：

```
typedef struct Object {
    /* ptr to class object */
    ClassObject *clazz;
    /*
     * A word containing either a "thin" lock or a "fat" monitor.  See
     * the comments in Sync.c for a description of its layout.
     */
    u4  lock;
} Object;
```

通过结构体 Object 定义了基本类的实现，这里有如下两个变量。

- lock：对应 Object 对象中的锁实现，即 notify wait 的处理。
- clazz：是结构体指针，姑且不看结构体内容，这里用了指针的定义。

下面会有更多的结构体定义：

```
struct DataObject {
    Object  obj;              /* MUST be first item */
    /* variable #of u4 slots; u8 uses 2 slots */
    u4      instanceData[1];
};
struct StringObject {
    Object  obj;              /* MUST be first item */
    /* variable #of u4 slots; u8 uses 2 slots */
    u4      instanceData[1];
};
```

我们看到最熟悉的一个词 StringObject，把这个结构体展开后，是下面的样子：

```
struct StringObject {
    /* ptr to class object */
    ClassObject  *clazz;
    /*
     * A word containing either a "thin" lock or a "fat" monitor.  See
     * the comments in Sync.c for a description of its layout.
     */
    u4          lock;
     /* variable #of u4 slots; u8 uses 2 slots */
    u4          instanceData[1];
};
```

由此不难发现，任何对象的内存结构体中第一行都是 Object 结构体，而这个结构体第一个总是一个 ClassObject，第二个总是 lock。按照 C++中的技巧，这些结构体可以当成 Object 结构体使用，因此所有的类在内存中都具有“对象”的功能，即可以找到一个类(ClassObject)，可以有一个锁(lock)。

StringObject 是对 String 类进行管理的数据对象，ArrayObejct 是数据相关的管理。

(2) ClassObject——Class 在加载后的表现形式

在解析完文件后，接下来，需要加载 Class 的具体内容。在 Dalvik 中，由数据结构 ClassObject 负责存放加载的信息。

如图 5-6 所示，加载过程会在内存中 alloc 几个区域，分别存放 directMethods、virtualMethods、sfields、ifields。这些信息是从 Dex 文件的数据区中读取的，首先会读取 Class 的详细信息，从中获得 directMethod、virtualMethod、sfield、ifield 等的信息，然后再读取。

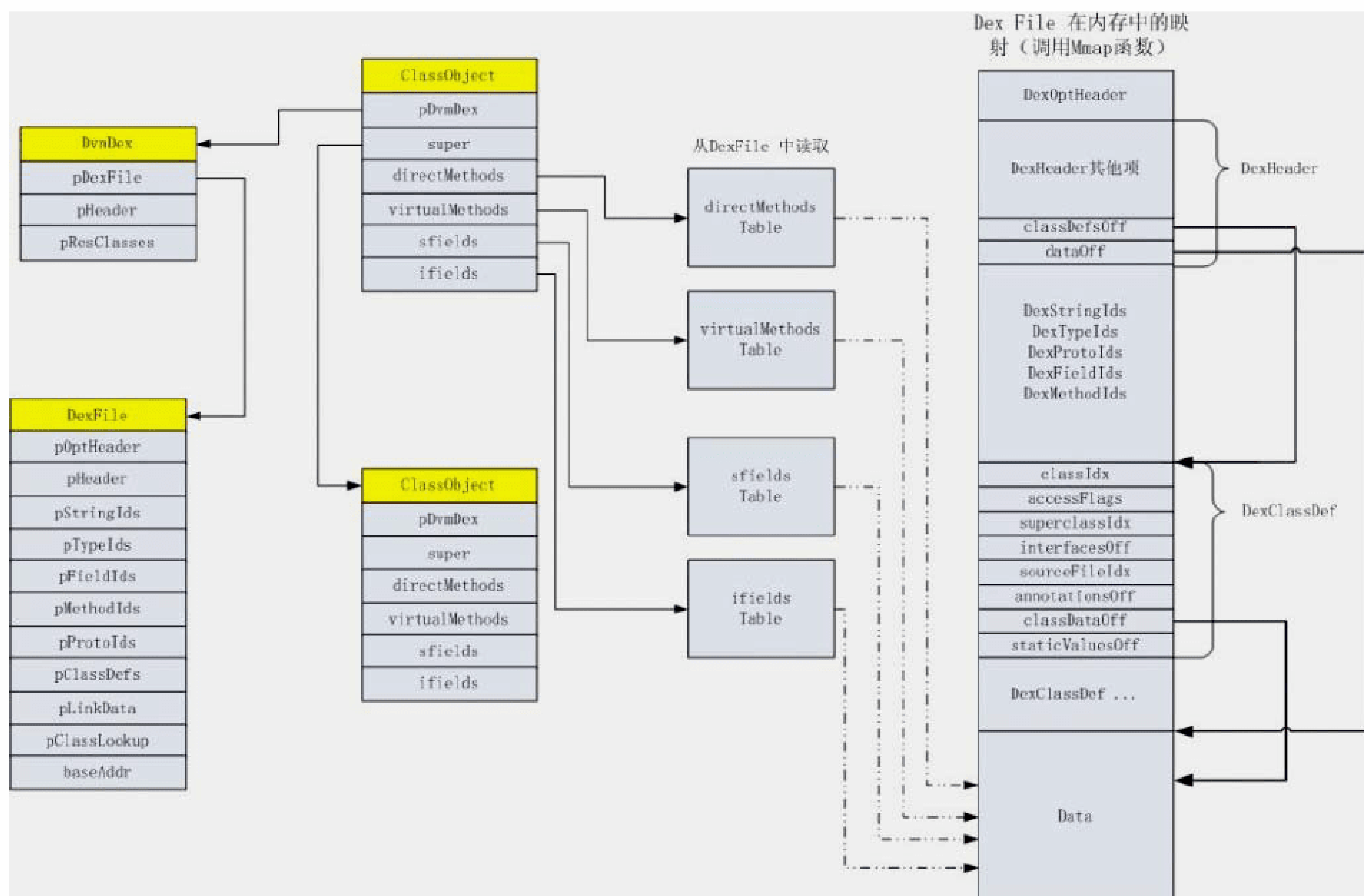


图 5-6　加载过程

在此需要注意，在 ClassObject 结构中有个名为 super 的成员，通过 super 成员，可以指向它的超类。

(3) 加载 Class 并生成相应 ClassObject 的函数

在讲解完加载数据结构的知识后，接下来开始分析负责加载工作的函数 findClassNoInit()。在获取 Class 索引时，会分为基本类库文件和用户类文件两种情况。在文件 grund.sh 中有如下语句：

```
export BOOTCLASSPATH=$bootpath/core.jar:$bootpath/ext.jar:$bootpath/
framework.jar:$bootpath/android.police.jar
```

上述语句指定了 Dalvik 所需的基本库文件，如果没有此语句，Dalvik 在启动过程中就会报错退出。

函数 LoadClassFromDex()会先读取 Class 的具体数据(从 ClassDataoff 处)，然后分别加载 directMethod、virtualMethod、ifield 和 sfield。

为了追求效率，在加载后需要将其缓存起来，以便以后使用方便。其次，在查找过程中，如果是顺序查找的话，会很慢，所以需要使用 gDvm.loadedClasses 这个 Hash 表来帮忙。如果一个子类需要调用超类的函数，那它当然要先加载超类了，可能的话，甚至会加载超类的超类。

接下来，使用 GDB 进行调试，在函数 findClassNoInit()处设置断点(在 GDB 提示符后输入“b findClassNoInit”)，在 GDB 提示符后连续几次执行“c”和“bt”。此时，可能会出现如下信息，在函数调用栈上可以多次看到 findClassNoInit()函数：

```
(gdb) bt
#0  findClassNoInit (descriptor=0xfef4c7f4 "??????%", loader=0x0, pDvmDex=0x0)
    at dalvik/vm/oo/Class.c:1373
#1  0xf6fc4d53 in dvmFindClassNoInit (descriptor=0xf5046a63 "Ljava/lang/Object;",
loader=0x0) at dalvik/vm/oo/Class.c:1194
#2  0xf6fc6c0a in dvmResolveClass (referrer=0xf5837400, classIdx=290,
    fromUnverifiedConstant=false) at dalvik/vm/oo/Resolve.c:94
#3  0xf6fc3476 in dvmLinkClass (clazz=0xf5837400, classesResolved=false)
    at dalvik/vm/oo/Class.c:2537
#4  0xf6fc1b67 in findClassNoInit (descriptor=0xf6ff0df6 "Ljava/lang/Class;",
loader=0x0, pDvmDex=0xa04c720) at dalvik/vm/oo/Class.c:1489
```

(4) 加载基本类库文件

接下来从另一个角度去观察，在文件 class.c 的 2575 行设置断点，然后等待程序停下。下面是 clazz 的内容：

```
(gdb) p clazz->super->descriptor
$6 = 0xf5046a63 "Ljava/lang/Object;"
 (gdb) p clazz->descriptor
$7 = 0xf5046121 "Ljava/lang/Class;"
```

然后先在 findClassNoInit()函数处设置断点，然后运行程序，并等待程序停下：

```
(gdb) b findClassNoInit
Breakpoint 2 at 0xf6fc13e0: file dalvik/vm/oo/Class.c, line 1373.
(gdb) c
Continuing.
```

看看究竟谁是第一个加载的 Class：

```
(gdb) bt
#0  findClassNoInit (descriptor=0x0, loader=0x0, pDvmDex=0x0)
    at dalvik/vm/oo/Class.c:1373
#1  0xf6fc32a1 in dvmLinkClass (clazz=0xf5837350, classesResolved=false)
    at dalvik/vm/oo/Class.c:2491
#2  0xf6fc1b67 in findClassNoInit (descriptor=0xf6ff1ded "Ljava/lang/Thread;",
loader=0x0, pDvmDex=0xa04c720) at dalvik/vm/oo/Class.c:1489
#3  0xf6f92692 in dvmThreadObjStartup () at dalvik/vm/Thread.c:328
#4  0xf6f800e6 in dvmStartup (argc=2, argv=0xa041190, ignoreUnrecognized=false,
pEnv=0xa0411a0) at dalvik/vm/Init.c:1155
```

```
#5  0xf6f8b8e3 in JNI_CreateJavaVM (p_vm=0xf6ff0df6, p_env=0xf6ff0df6,
vm args=0xfef4d0b0) at dalvik/vm/Jni.c:4198
#6  0x08048893 in main (argc=3, argv=0xfef4d168) at dalvik/dalvikvm/Main.c:212
```

由上述函数的调用顺序可得出：

```
main → JNI_CreateJavaVM → dvmStartup → dvmThreadObjStartup
 → dvmFindSystemClassNoInit → findClassNoInit
```

在上述调用栈中没有 dvmFindSystemClassNoInit，是因为编译器将其作为 inline 优化了，导致 GDB 看不到有 dvmFindSystemClassNoInit 的栈。但是不要担心，我们可以从回溯栈中看到 dvmFindSystemClassNoInit。

(5) 加载用户类文件

在加载用户类文件时，会先加载一个 Class，然后这个 Class 去负责用户类文件的加载，而这个 Class 又会通过 JNI 的方式去调用 findClassNoInit。具体加载过程与前面介绍的基本类库加载类似，读者可以参考前面的知识来理解。在此为节省本书篇幅，不再详细介绍。

5.3 Dalvik VM的内存系统

内存管理是 Dalvik VM 中的一个重要组件，其内存管理的核心是分别实现内存分配和回收工作。Java 语言使用 new 操作符来分配内存，但是 Java 语言并没有提供任何操作来释放内存，而是通过垃圾收集机制来回收内存。对于内存管理的实现，我们通过如下三个方面来加以分析：

- 内存分配。
- 内存回收。
- 内存管理和调试。

在本节的内容中，将详细讲解 Dalvik VM 的内存系统。

5.3.1 如何分配内存

在本节的内容中，将分析 Dalvik 虚拟机是如何分配内存的。

(1) 对象布局

内存管理的主要操作之一是为 Java 对象分配内存，Java 对象在虚拟机中的内存布局如图 5-7 所示。

所有的对象都有一个相同的头部 clazz 和 lock。

- clazz：指向该对象的类对象，类对象用来描述该对象所属的类，这样可以很容易从一个对象获取该对象所属的类的具体信息。
- lock：是一个无符号整数，用以实现对象的同步。
- data：用于存放对象数据，根据对象的不同，数据区的大小是不同的。

(2) 堆

堆是 Dalvik 虚拟机从操作系统分配的一块连续的虚拟内存。如图 5-8 所示。其中 heapBase 表示堆的起始地址，heapLimit 表示堆的最大地址，堆大小的最大值可以通过“-Xmx”选项或

dalvik.vm.heapsize 指定。在原生系统中，一般 dalvik.vm.heapsize 值是 32MB，在 MIUI 中，我们将其设为 64MB。

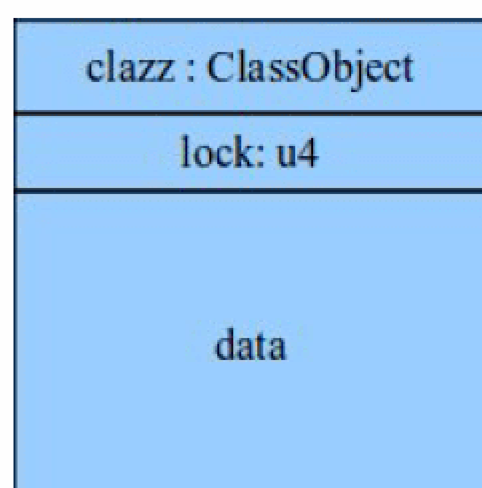


图 5-7　内存布局

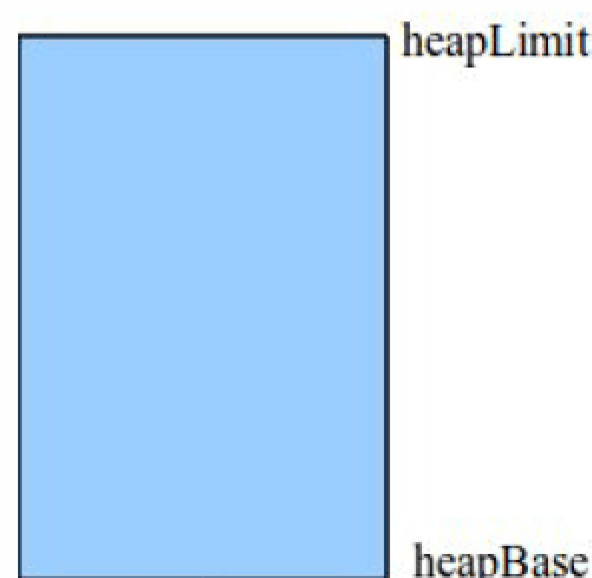


图 5-8　堆结构

(3)　堆内存位图

在虚拟机中维护了两个对应于堆内存的位图，称为 liveBits 和 markBits。在对象布局中，我们看到对象最小占用 8 个字节。在为对象分配内存时，要求必须 8 字节对齐。这也就是说，对象的大小会调整为 8 字节的倍数。比如说，一个对象的实际大小是 13 字节，但是在分配内存的时候分配 16 字节。因此所有对象的起始地址一定是 8 字节的倍数。堆内存位图就是用来描述堆内存的，每一个 bit 描述 8 个字节，因此堆内存位图的大小是对堆的 1/64。对于 MIUI 的实现来说，这两个位图各占 1MB。

liveBits 的功能是跟踪堆中已经分配的内存，每分配一个对象时，对象的内存起始地址对应于位图中的位被设为 1。在介绍垃圾收集时我们会进一步地分析 liveBits 和 markBits 这两个位图的作用。

(4)　堆的内存管理

在 Dalvik 虚拟机的实现中，是通过底层的 bionicC 库的 malloc/free 操作来分配/释放内存的。库 bionicC 的 malloc/free 操作是基于 DougLea 的实现(dlmalloc)，这是一个被广泛使用、久经考验的 C 内存管理库。有关库 dlmalloc 的基本知识，读者可以参阅相关的资料。

(5)　dvmAllocObject

在 Dalvik 虚拟机中，操作符 new 最终对应 C 函数 dvmAllocObject()。下面通过伪码的形式列出 dvmAllocObject 的实现：

```
Object* dvmAllocObject(ClassObject *clazz, int flags) {
    n = get object size form class object clazz
    first try: allocate n bytes from heap
    if first try failed {
        run garbage collector without collecting soft references
        second try: allocate n bytes from heap
    }
    if second try failed {
        third try: grow the heap and allocate n bytes from heap
        //堆是虚拟内存，一开始并未分配所有的物理内存，只要还没有达到虚拟内存的最大值，
        //就可以通过获取更多物理内存的方式来扩展堆
    }
    if third try failed {
```

```
        run garbage collector with collecting soft references
        fourth try: grow the hap and allocate n bytes from heap
    }
    if fourth try failed, return null pointer, dalvik vm will abort
}
```

由此可以看出，为了分配内存，虚拟机尽了最大的努力，做了 4 次尝试。其中进行了两次垃圾收集，第一次不收集 SoftReference，第二次收集 SoftReference。从中我们也可以看出垃圾收集的时机，实质上，在 Dalvik 虚拟机实现中有 3 个时机可以触发垃圾收集的运行：

- 程序员显式调用 System.gc()。
- 内存分配失败时。
- 如果分配的对象大小超过 384KB，运行并发标记(Concurrent Mark)。

综上所述，在 Dalvik 虚拟机中，内存分配操作的流程相对比较简单、直观，从一个堆中分配可用内存，分配失败时会触发垃圾收集。

5.3.2　分析内存管理机制的源码

Dalvik 虚拟机的内存管理需要依赖于 Linux 的内存管理机制，Dalvik 的内存管理的实现源码保存在 vm\alloc 目录中。在接下来的内容中，将通过对源码的分析来简要讲解 Dalvik 内存管理机制的基本知识。

(1)　表示堆的结构体

在文件 HeapSource.c 中定义表示堆的结构体，其源码如下所示：

```
typedef struct {
    /*使用 dlmalloc 分配的内存
     */
    mspace *msp;

    HeapBitmap objectBitmap;

    /* 堆可以增长的最大值
     */
    size_t absoluteMaxSize;

    /* 已分配的字节数
     */
    size t bytesAllocated;

    /* 已分配的对象数
     */
    size_t objectsAllocated;
} Heap;
```

(2)　表示位图堆的结构体数据

在文件 HeapBitmap.h 中定义表示位图堆的结构体数据，其源码如下所示：

```
typedef struct {
```

```
    /* 位图数据
     */
    unsigned long int *bits;

    /* 位图的大小
     */
    size_t bitsLen;

    /* 位图对应的对象指针数组的首地址
     */
    uintptr_t base;

    /* 位图使用中的最后一位被设置的对象指针地址，如果全没设置则(max < base)
     */
    uintptr_t max;
} HeapBitmap;
```

(3) HeapSource 结构体

在 Dalvik 虚拟机中，使用结构体 HeapSource 来管理各种 Heap 数据，Heap 只是其中的一个子项，其源码在文件 HeapSource.c 中定义：

```
struct HeapSource {
    /* 堆的使用率，范围从 1 到 HEAP_UTILIZATION_MAX
     */
    size_t targetUtilization;

    /* 分配堆的最小尺寸
     */
    size_t minimumSize;

    /* 堆分配的初始尺寸
     */
    size_t startSize;

    /* 允许分配的堆增长到的最大尺寸
     */
    size_t absoluteMaxSize;

    /* 理想的堆的最大尺寸
     */
    size_t idealSize;

    /* 在垃圾收集前允许堆分配的最大尺寸
     */
    size_t softLimit;

    /* 堆数组，最大尺寸为 3
     */
    Heap heaps[HEAP_SOURCE_MAX_HEAP_COUNT];
```

```
    /* 当前堆的个数
     */
    size_t numHeaps;

    /* 对外分配计数
     */
    size_t externalBytesAllocated;

    /* 允许外部分配的最大值
     */
    size_t externalLimit;

    /* 在创建这个 HeapSource 的时候是否是 Zygote 模式，确定是否有 Zygote 进程
     */
    bool sawZygote;
};
```

(4)　与 mark bits 相关的结构体

在文件 MarkSweep.h 中定义了与 mark bits 相关的结构体，其源码如下所示：

```
typedef struct {
    /* 允许增长到的最低地址
     */
    const Object **limit;

    /* 栈顶
     */
    const Object **top;

    /* 栈底
     */
    const Object **base;
} GcMarkStack;

typedef struct {
    /* 存放位图的数组
     */
    HeapBitmap bitmaps[HEAP_SOURCE_MAX_HEAP_COUNT];
    /* 位图数
     */
    size_t numBitmaps;
    /* GC 标记栈
     */
    GcMarkStack stack;
    /* 存放地址上限的标志
     */
    const void *finger;   // only used while scanning/recursing.
} GcMarkContext;
```

(5) 结构体 GcHeap

在文件 HeapInternal.h 中定义了 Dalvik 的垃圾回收机制，需要用到结构体 GcHeap，其源码如下所示：

```
struct GcHeap {
    /* HeapSource 结构，包含了所有的堆数据 */
    HeapSource      *heapSource;

    /* 存储不能被垃圾回收对象的参考表
     */
    HeapRefTable    nonCollectableRefs;

    /*存储一些当被垃圾回收时需要执行 finalization 方法的参考表
     *
     */
    LargeHeapRefTable  *finalizableRefs;

    /*存储需要执行 finalization 方法的对象的参考表
     */
    LargeHeapRefTable  *pendingFinalizationRefs;

    /* 软引用对象列表
     */
    Object          *softReferences;
    /* 弱引用对象列表
     */
    Object          *weakReferences;
    /* 影子引用对象列表
     */
    Object          *phantomReferences;

    /* 需要被执行 clear 或 enqueue 方法的引用对象列表
     */
    LargeHeapRefTable  *referenceOperations;

    /* 如果对象不为空，则表示 HeapWorker 线程正在执行
     * executing.
     */
    Object *heapWorkerCurrentObject;
    Method *heapWorkerCurrentMethod;

    /*如果 heapWorkerCurrentObject 不为空，表示 HeapWorker 开始执行这个方法的时间
     */
    u8 heapWorkerInterpStartTime;
    /*如果 heapWorkerCurrentObject 不为空，表示 HeapWorker  CPU 开始执行这个方法的时间
     */
    u8 heapWorkerInterpCpuStartTime;
```

```
    /* 下一次裁剪 Heap Source 的时间
     */
    struct timespec heapWorkerNextTrim;

    /* 标记步骤中的状态
     */
    GcMarkContext   markContext;

    /* GC 开始的时间.
     */
    u8   gcStartTime;

    /* 是否正在执行 GC
     */
    bool  gcRunning;

    /* GC 时引用对象回收多少，分为以下三种情况
     * 不回收，回收一半，全部回收
     */
    enum { SR_COLLECT_NONE, SR_COLLECT_SOME, SR_COLLECT_ALL }
           softReferenceCollectionState;

    /* 存在多少软引用对象时开始回收引用对象
     */
    size_t         softReferenceHeapSizeThreshold;

    /*当软引用回收策略为回收一半时使用的概率值
     */
    int            softReferenceColor;

    /* 引用收集策略
     */
    bool           markAllReferents;
#if DVM_TRACK_HEAP_MARKING
    /* Every time an unmarked object becomes marked, markCount
     * is incremented and markSize increases by the size of
     * that object.
     */
    size_t         markCount;
    size_t         markSize;
#endif

    /* 下面是与调试相关的信息
     */
    int            ddmHpifWhen;
    int            ddmHpsgWhen;
    int            ddmHpsgWhat;
    int            ddmNhsgWhen;
```

```
    int            ddmNhsgWhat;

#if WITH_HPROF
    bool           hprofDumpOnGc;
    const char    *hprofFileName;
    hprof_context_t *hprofContext;
    int            hprofResult;
#endif
};
```

(6) 初始化垃圾回收器

在文件 Init.c 中，通过函数 dvmGcStartup()来初始化垃圾回收器：

```
bool dvmGcStartup(void)
{
    dvmInitMutex(&gDvm.gcHeapLock);
    return dvmHeapStartup();
}
```

(7) 初始化与 Heap 相关的信息

在文件 alloc\Heap.c 中，通过 dvmHeapStartup()函数来初始化和 Heap 相关的信息，例如常见的内存分配和内存管理等工作。其源码如下所示：

```
bool dvmHeapStartup()
{
    GcHeap *gcHeap;
#if defined(WITH_ALLOC_LIMITS)
    gDvm.checkAllocLimits = false;
    gDvm.allocationLimit = -1;
#endif
    gcHeap = dvmHeapSourceStartup(gDvm.heapSizeStart, gDvm.heapSizeMax);
    if (gcHeap == NULL) {
        return false;
    }
    gcHeap->heapWorkerCurrentObject = NULL;
    gcHeap->heapWorkerCurrentMethod = NULL;
    gcHeap->heapWorkerInterpStartTime = 0LL;
    gcHeap->softReferenceCollectionState = SR_COLLECT_NONE;
    gcHeap->softReferenceHeapSizeThreshold = gDvm.heapSizeStart;
    gcHeap->ddmHpifWhen = 0;
    gcHeap->ddmHpsgWhen = 0;
    gcHeap->ddmHpsgWhat = 0;
    gcHeap->ddmNhsgWhen = 0;
    gcHeap->ddmNhsgWhat = 0;
#if WITH_HPROF
    gcHeap->hprofDumpOnGc = false;
    gcHeap->hprofContext = NULL;
#endif
    /* This needs to be set before we call dvmHeapInitHeapRefTable().
```

```
    */
    gDvm.gcHeap = gcHeap;
    /* Set up the table we'll use for ALLOC_NO_GC.
     */
    if (!dvmHeapInitHeapRefTable(&gcHeap->nonCollectableRefs,
                          kNonCollectableRefDefault))
    {
        LOGE_HEAP("Can't allocate GC_NO_ALLOC table/n");
        goto fail;
    }
    /* Set up the lists and lock we'll use for finalizable
     * and reference objects.
     */
    dvmInitMutex(&gDvm.heapWorkerListLock);
    gcHeap->finalizableRefs = NULL;
    gcHeap->pendingFinalizationRefs = NULL;
    gcHeap->referenceOperations = NULL;
    /* Initialize the HeapWorker locks and other state
     * that the GC uses.
     */
    dvmInitializeHeapWorkerState();
    return true;
fail:
    gDvm.gcHeap = NULL;
    dvmHeapSourceShutdown(gcHeap);
    return false;
}
```

(8)　创建 GcHeap

在文件 alloc\HeapSource.c 中，通过函数 dvmHeapSourceStartup()来创建 GcHeap。其源码如下所示：

```
GcHeap* dvmHeapSourceStartup(size t startSize, size t absoluteMaxSize)
{
    GcHeap *gcHeap;
    HeapSource *hs;
    Heap *heap;
    mspace msp;

    assert(gHs == NULL);

    if (startSize > absoluteMaxSize) {
        LOGE("Bad heap parameters (start=%d, max=%d)\n",
           startSize, absoluteMaxSize);
        return NULL;
    }

    /* Create an unlocked dlmalloc mspace to use as
     * the small object heap source.
```

```
     */
    msp = createMspace(startSize, absoluteMaxSize, 0);
    if (msp == NULL) {
        return false;
    }

    /* Allocate a descriptor from the heap we just created.
     */
    gcHeap = mspace_malloc(msp, sizeof(*gcHeap));
    if (gcHeap == NULL) {
        LOGE_HEAP("Can't allocate heap descriptor\n");
        goto fail;
    }
    memset(gcHeap, 0, sizeof(*gcHeap));

    hs = mspace_malloc(msp, sizeof(*hs));
    if (hs == NULL) {
        LOGE_HEAP("Can't allocate heap source\n");
        goto fail;
    }
    memset(hs, 0, sizeof(*hs));

    hs->targetUtilization = DEFAULT_HEAP_UTILIZATION;
    hs->minimumSize = 0;
    hs->startSize = startSize;
    hs->absoluteMaxSize = absoluteMaxSize;
    hs->idealSize = startSize;
    hs->softLimit = INT_MAX;    // no soft limit at first
    hs->numHeaps = 0;
    hs->sawZygote = gDvm.zygote;
    if (!addNewHeap(hs, msp, absoluteMaxSize)) {
        LOGE_HEAP("Can't add initial heap\n");
        goto fail;
    }

    gcHeap->heapSource = hs;

    countAllocation(hs2heap(hs), gcHeap, false);
    countAllocation(hs2heap(hs), hs, false);

    gHs = hs;
    return gcHeap;

fail:
    destroy_contiguous_mspace(msp);
    return NULL;
}
```

(9)　追踪位置

在文件 alloc\HeapSource.c 中，通过函数 countAllocation()在 Heap::obiect Bitmap 上进行标记，以便追踪这些区域的位置。其源码如下所示：

```
static inline void countAllocation(Heap *heap, const void *ptr, bool isObj)
{
    assert(heap->bytesAllocated < mspace_footprint(heap->msp));

    heap->bytesAllocated +=
      mspace_usable_size(heap->msp, ptr) + HEAP_SOURCE_CHUNK_OVERHEAD;
    if (isObj) {
        heap->objectsAllocated++;
        //标记回收
        dvmHeapBitmapSetObjectBit(&heap->objectBitmap, ptr);
    }
    assert(heap->bytesAllocated < mspace_footprint(heap->msp));
}
HB_INLINE_PROTO(
    bool
    dvmHeapBitmapMayContainObject(const HeapBitmap *hb, const void *obj)
)
{
    const uintptr_t p = (const uintptr_t)obj;

    assert((p & (HB_OBJECT_ALIGNMENT - 1)) == 0);

    return p >= hb->base && p <= hb->max;
}
HB_INLINE_PROTO(
    bool
    dvmHeapBitmapCoversAddress(const HeapBitmap *hb, const void *obj)
)
{
    assert(hb != NULL);

    if (obj != NULL) {
        const uintptr_t offset = (uintptr_t)obj - hb->base;
        const size_t index = HB_OFFSET_TO_INDEX(offset);
        return index < hb->bitsLen / sizeof(*hb->bits);
    }
    return false;
}
...
```

(10) 分配空间

在文件 Heap.c 中，通过函数 dvmMalloc()实现空间的分配工作。其源码如下所示：

```
void* dvmMalloc(size_t size, int flags)
{
```

```
    GcHeap *gcHeap = gDvm.gcHeap;
    DvmHeapChunk *hc;
    void *ptr;
    bool triedGc, triedGrowing;

#if 0
    /* handy for spotting large allocations */
    if (size >= 100000) {
        LOGI("dvmMalloc(%d):\n", size);
        dvmDumpThread(dvmThreadSelf(), false);
    }
#endif

#if defined(WITH_ALLOC_LIMITS)
    /*
     * See if they've exceeded the allocation limit for this thread.
     *
     * A limit value of -1 means "no limit".
     *
     * This is enabled at compile time because it requires us to do a
     * TLS lookup for the Thread pointer.  This has enough of a performance
     * impact that we don't want to do it if we don't have to.  (Now that
     * we're using gDvm.checkAllocLimits we may want to reconsider this,
     * but it's probably still best to just compile the check out of
     * production code -- one less thing to hit on every allocation.)
     */
    if (gDvm.checkAllocLimits) {
        Thread *self = dvmThreadSelf();
        if (self != NULL) {
            int count = self->allocLimit;
            if (count > 0) {
                self->allocLimit--;
            } else if (count == 0) {
                /* fail! */
                assert(!gDvm.initializing);
                self->allocLimit = -1;
                dvmThrowException("Ldalvik/system/AllocationLimitError;",
                    "thread allocation limit exceeded");
                return NULL;
            }
        }
    }

    if (gDvm.allocationLimit >= 0) {
        assert(!gDvm.initializing);
        gDvm.allocationLimit = -1;
        dvmThrowException("Ldalvik/system/AllocationLimitError;",
            "global allocation limit exceeded");
        return NULL;
```

```
    }
#endif

    dvmLockHeap();

    /* Try as hard as possible to allocate some memory.
     */
    hc = tryMalloc(size);
    if (hc != NULL) {
alloc_succeeded:
        /* We've got the memory.
         */
        if ((flags & ALLOC_FINALIZABLE) != 0) {
            /* This object is an instance of a class that
             * overrides finalize().  Add it to the finalizable list.
             *
             * Note that until DVM_OBJECT_INIT() is called on this
             * object, its clazz will be NULL.  Since the object is
             * in this table, it will be scanned as part of the root
             * set.  scanObject() explicitly deals with the NULL clazz.
             */
            if (!dvmHeapAddRefToLargeTable(&gcHeap->finalizableRefs,
                                    (Object *)hc->data))
            {
                LOGE_HEAP("dvmMalloc(): no room for any more "
                        "finalizable objects\n");
                dvmAbort();
            }
        }

#if WITH_OBJECT_HEADERS
        hc->header = OBJECT_HEADER;
        hc->birthGeneration = gGeneration;
#endif
        ptr = hc->data;

        /* The caller may not want us to collect this object.
         * If not, throw it in the nonCollectableRefs table, which
         * will be added to the root set when we GC.
         *
         * Note that until DVM_OBJECT_INIT() is called on this
         * object, its clazz will be NULL.  Since the object is
         * in this table, it will be scanned as part of the root
         * set.  scanObject() explicitly deals with the NULL clazz.
         */
        if ((flags & ALLOC_NO_GC) != 0) {
            if (!dvmHeapAddToHeapRefTable(&gcHeap->nonCollectableRefs, ptr)) {
                LOGE_HEAP("dvmMalloc(): no room for any more "
                        "ALLOC_NO_GC objects: %zd\n",
```

```
                    dvmHeapNumHeapRefTableEntries(
                            &gcHeap->nonCollectableRefs));
                dvmAbort();
            }
        }

#ifdef WITH_PROFILER
        if (gDvm.allocProf.enabled) {
            Thread *self = dvmThreadSelf();
            gDvm.allocProf.allocCount++;
            gDvm.allocProf.allocSize += size;
            if (self != NULL) {
                self->allocProf.allocCount++;
                self->allocProf.allocSize += size;
            }
        }
#endif
    } else {
        /* The allocation failed.
         */
        ptr = NULL;

#ifdef WITH_PROFILER
        if (gDvm.allocProf.enabled) {
            Thread *self = dvmThreadSelf();
            gDvm.allocProf.failedAllocCount++;
            gDvm.allocProf.failedAllocSize += size;
            if (self != NULL) {
                self->allocProf.failedAllocCount++;
                self->allocProf.failedAllocSize += size;
            }
        }
#endif
    }

    dvmUnlockHeap();

    if (ptr != NULL) {
        /*
         * If this block is immediately GCable, and they haven't asked us not
         * to track it, add it to the internal tracking list.
         *
         * If there's no "self" yet, we can't track it.  Calls made before
         * the Thread exists should use ALLOC_NO_GC.
         */
        if ((flags & (ALLOC_DONT_TRACK | ALLOC_NO_GC)) == 0) {
            dvmAddTrackedAlloc(ptr, NULL);
        }
```

```
    } else {
        /*
         * The allocation failed; throw an OutOfMemoryError.
         */
        throwOOME();
    }
    return ptr;
}
```

上述具体分配过程由 tryMalloc 函数控制，具体执行流程为：

```
tryMalloc() → gcForMalloc() → dvmCollectGarbageInternal()
```

当满足条件时，会调用 dvmCollectGarbageInternal()函数进行垃圾回收。

5.4　分析Dalvik VM的启动过程

经过本章前面内容的学习可知，Android 系统中的应用程序进程是由 Zygote 进程孕育出来的，而 Zygote 进程又是由 Init 进程启动的。在启动 Zygote 进程时，会创建一个 Dalvik VM 实例，每当 Zygote 进程孕育出一个新的应用程序进程时，会将这个 Dalvik VM 实例复制到新的应用程序中，这样可以使每个应用程序进程都拥有一个独立的 Dalvik VM 实例。在本节的内容中，将详细分析启动 Dalvik VM 的源码，更加深入地了解启动 Dalvik VM 的过程。

5.4.1　创建一个Dalvik VM实例

在启动 Zygote 进程的过程中，以调用类 AndroidRuntime 的成员函数 start 作为启动 Dalvik VM 的第一步。在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中，调用函数 start 作为启动 Dalvik VM 的第一步。函数 start 调用函数 startVm 创建一个 Dalvik VM 实例，并且保存在成员变量 mJavaVM 中。然后调用成员函数 startReg 注册 Android 核心类的 JNI 方法，并调用参数 className 所对应的一个 Java 函数 main 作为 Zygote 进程的 Java 层入口。函数 start 的具体实现代码如下所示：

```
void AndroidRuntime::start(const char *className, const bool startSystemServer)
{
    /* 函数 startVm 用于创建一个 Dalvik 虚拟机实例，并且保存在成员变量 mJavaVM 中 */
    if (startVm(&mJavaVM, &env) != 0)
        goto bail;
    /*
     *函数 startReg 用于注册 Android 核心类的 JNI 方法
     */
    if (startReg(env) < 0) {
        LOGE("Unable to register all android natives\n");
        goto bail;
    }
```

```
    ...

    /*
     * 开始启动虚拟机
     */
    jclass startClass;
    jmethodID startMeth;

    slashClassName = strdup(className);
    for (cp=slashClassName; *cp!='\0'; cp++)
        if (*cp == '.')
            *cp = '/';

    startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        LOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            LOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
            ...
        }
    }

    LOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        LOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        LOGW("Warning: VM did not shut down cleanly\n");

    ...
}
```

在退出上述函数代码之前，需要调用刚刚创建的 Dalvik VM 实例的如下两个成员函数。

- DetachCurrentThread：功能是将 Zygote 进程的主线程脱离刚刚创建的 Dalvik 虚拟机实例。
- DestroyJavaVM：用于销毁刚刚创建的 Dalvik VM 实例。

5.4.2 指定控制选项

函数 startVm 在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，功能是在启动 Dalvik VM 时指定一系列有用的控制选项。函数 startVm 的具体实现代码如下所示：

```
int AndroidRuntime::startVm(JavaVM **pJavaVM, JNIEnv **pEnv)
{
    int result = -1;
    JavaVMInitArgs initArgs;
    JavaVMOption opt;
    char propBuf[PROPERTY_VALUE_MAX];
    char stackTraceFileBuf[PROPERTY_VALUE_MAX];
    char dexoptFlagsBuf[PROPERTY_VALUE_MAX];
    char enableAssertBuf[sizeof("-ea:")-1 + PROPERTY_VALUE_MAX];
    char jniOptsBuf[sizeof("-Xjniopts:")-1 + PROPERTY_VALUE_MAX];
    char heapstartsizeOptsBuf[sizeof("-Xms")-1 + PROPERTY_VALUE_MAX];
    char heapsizeOptsBuf[sizeof("-Xmx")-1 + PROPERTY_VALUE_MAX];
    char heapgrowthlimitOptsBuf[
      sizeof("-XX:HeapGrowthLimit=")-1 + PROPERTY_VALUE_MAX];
    char heapminfreeOptsBuf[sizeof("-XX:HeapMinFree=")-1 + PROPERTY_VALUE_MAX];
    char heapmaxfreeOptsBuf[sizeof("-XX:HeapMaxFree=")-1 + PROPERTY_VALUE_MAX];
    char heaptargetutilizationOptsBuf[
      sizeof("-XX:HeapTargetUtilization=")-1 + PROPERTY_VALUE_MAX];
    char extraOptsBuf[PROPERTY_VALUE_MAX];
    char *stackTraceFile = NULL;
    bool checkJni = false;
    bool checkDexSum = false;
    bool logStdio = false;
    enum {
      kEMDefault,
      kEMIntPortable,
      kEMIntFast,
      kEMJitCompiler,
    } executionMode = kEMDefault;

    property_get("dalvik.vm.checkjni", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkJni = true;
    } else if (strcmp(propBuf, "false") != 0) {
        /* property is neither true nor false; fall back on kernel parameter */
        property_get("ro.kernel.android.checkjni", propBuf, "");
        if (propBuf[0] == '1') {
            checkJni = true;
        }
    }

    property_get("dalvik.vm.execution-mode", propBuf, "");
    if (strcmp(propBuf, "int:portable") == 0) {
        executionMode = kEMIntPortable;
    } else if (strcmp(propBuf, "int:fast") == 0) {
        executionMode = kEMIntFast;
    } else if (strcmp(propBuf, "int:jit") == 0) {
        executionMode = kEMJitCompiler;
    }
```

```
property get("dalvik.vm.stack-trace-file", stackTraceFileBuf, "");

property_get("dalvik.vm.check-dex-sum", propBuf, "");
if (strcmp(propBuf, "true") == 0) {
    checkDexSum = true;
}

property_get("log.redirect-stdio", propBuf, "");
if (strcmp(propBuf, "true") == 0) {
    logStdio = true;
}

strcpy(enableAssertBuf, "-ea:");
property_get("dalvik.vm.enableassertions", enableAssertBuf+4, "");

strcpy(jniOptsBuf, "-Xjniopts:");
property_get("dalvik.vm.jniopts", jniOptsBuf+10, "");

/* route exit() to our handler */
opt.extraInfo = (void*)runtime exit;
opt.optionString = "exit";
mOptions.add(opt);

/* route fprintf() to our handler */
opt.extraInfo = (void*)runtime_vfprintf;
opt.optionString = "vfprintf";
mOptions.add(opt);

/* register the framework-specific "is sensitive thread" hook */
opt.extraInfo = (void*)runtime_isSensitiveThread;
opt.optionString = "sensitiveThread";
mOptions.add(opt);

opt.extraInfo = NULL;

/* enable verbose; standard options are { jni, gc, class } */
//options[curOpt++].optionString = "-verbose:jni";
opt.optionString = "-verbose:gc";
mOptions.add(opt);
//options[curOpt++].optionString = "-verbose:class";

/*
 * The default starting and maximum size of the heap.  Larger
 * values should be specified in a product property override.
 */
strcpy(heapstartsizeOptsBuf, "-Xms");
property get("dalvik.vm.heapstartsize", heapstartsizeOptsBuf+4, "4m");
opt.optionString = heapstartsizeOptsBuf;
```

```
mOptions.add(opt);
strcpy(heapsizeOptsBuf, "-Xmx");
property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
opt.optionString = heapsizeOptsBuf;
mOptions.add(opt);

// Increase the main thread's interpreter stack size for bug 6315322.
opt.optionString = "-XX:mainThreadStackSize=24K";
mOptions.add(opt);

strcpy(heapgrowthlimitOptsBuf, "-XX:HeapGrowthLimit=");
property_get("dalvik.vm.heapgrowthlimit", heapgrowthlimitOptsBuf+20, "");
if (heapgrowthlimitOptsBuf[20] != '\0') {
    opt.optionString = heapgrowthlimitOptsBuf;
    mOptions.add(opt);
}

strcpy(heapminfreeOptsBuf, "-XX:HeapMinFree=");
property_get("dalvik.vm.heapminfree", heapminfreeOptsBuf+16, "");
if (heapminfreeOptsBuf[16] != '\0') {
    opt.optionString = heapminfreeOptsBuf;
    mOptions.add(opt);
}

strcpy(heapmaxfreeOptsBuf, "-XX:HeapMaxFree=");
property_get("dalvik.vm.heapmaxfree", heapmaxfreeOptsBuf+16, "");
if (heapmaxfreeOptsBuf[16] != '\0') {
    opt.optionString = heapmaxfreeOptsBuf;
    mOptions.add(opt);
}

strcpy(heaptargetutilizationOptsBuf, "-XX:HeapTargetUtilization=");
property_get(
  "dalvik.vm.heaptargetutilization", heaptargetutilizationOptsBuf+26, "");
if (heaptargetutilizationOptsBuf[26] != '\0') {
    opt.optionString = heaptargetutilizationOptsBuf;
    mOptions.add(opt);
}

/*
 * Enable or disable dexopt features, such as bytecode verification and
 * calculation of register maps for precise GC.
 */
property_get("dalvik.vm.dexopt-flags", dexoptFlagsBuf, "");
if (dexoptFlagsBuf[0] != '\0') {
    const char *opc;
    const char *val;

    opc = strstr(dexoptFlagsBuf, "v=");     /* verification */
```

```
        if (opc != NULL) {
            switch (*(opc+2)) {
            case 'n':   val = "-Xverify:none";     break;
            case 'r':   val = "-Xverify:remote";   break;
            case 'a':   val = "-Xverify:all";      break;
            default:    val = NULL;                break;
            }

            if (val != NULL) {
                opt.optionString = val;
                mOptions.add(opt);
            }
        }

        opc = strstr(dexoptFlagsBuf, "o=");    /* optimization */
        if (opc != NULL) {
            switch (*(opc+2)) {
            case 'n':   val = "-Xdexopt:none";     break;
            case 'v':   val = "-Xdexopt:verified";  break;
            case 'a':   val = "-Xdexopt:all";      break;
            case 'f':   val = "-Xdexopt:full";     break;
            default:    val = NULL;                break;
            }

            if (val != NULL) {
                opt.optionString = val;
                mOptions.add(opt);
            }
        }

        opc = strstr(dexoptFlagsBuf, "m=y");   /* register map */
        if (opc != NULL) {
            opt.optionString = "-Xgenregmap";
            mOptions.add(opt);

            /* turn on precise GC while we're at it */
            opt.optionString = "-Xgc:precise";
            mOptions.add(opt);
        }
    }

    /* enable debugging; set suspend=y to pause during VM init */
    /* use android ADB transport */
    opt.optionString =
        "-agentlib:jdwp=transport=dt_android_adb,suspend=n,server=y";
    mOptions.add(opt);

    ALOGD("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
```

```
        /* extended JNI checking */
        opt.optionString = "-Xcheck:jni";
        mOptions.add(opt);

        /* set a cap on JNI global references */
        opt.optionString = "-Xjnigreflimit:2000";
        mOptions.add(opt);

        /* with -Xcheck:jni, this provides a JNI function call trace */
        //opt.optionString = "-verbose:jni";
        //mOptions.add(opt);
    }

    char lockProfThresholdBuf[sizeof("-Xlockprofthreshold:") + sizeof(propBuf)];
    property_get("dalvik.vm.lockprof.threshold", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(lockProfThresholdBuf, "-Xlockprofthreshold:");
        strcat(lockProfThresholdBuf, propBuf);
        opt.optionString = lockProfThresholdBuf;
        mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected opcodes. Eg "1-0a,3c,f1-ff" */
    char jitOpBuf[sizeof("-Xjitop:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.op", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitOpBuf, "-Xjitop:");
        strcat(jitOpBuf, propBuf);
        opt.optionString = jitOpBuf;
        mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected methods */
    char jitMethodBuf[sizeof("-Xjitmethod:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.method", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitMethodBuf, "-Xjitmethod:");
        strcat(jitMethodBuf, propBuf);
        opt.optionString = jitMethodBuf;
        mOptions.add(opt);
    }

    if (executionMode == kEMIntPortable) {
        opt.optionString = "-Xint:portable";
        mOptions.add(opt);
    } else if (executionMode == kEMIntFast) {
        opt.optionString = "-Xint:fast";
        mOptions.add(opt);
    } else if (executionMode == kEMJitCompiler) {
```

```
        opt.optionString = "-Xint:jit";
        mOptions.add(opt);
    }

    if (checkDexSum) {
        /* perform additional DEX checksum tests */
        opt.optionString = "-Xcheckdexsum";
        mOptions.add(opt);
    }

    if (logStdio) {
        /* convert stdout/stderr to log messages */
        opt.optionString = "-Xlog-stdio";
        mOptions.add(opt);
    }

    if (enableAssertBuf[4] != '\0') {
        /* accept "all" to mean "all classes and packages" */
        if (strcmp(enableAssertBuf+4, "all") == 0)
            enableAssertBuf[3] = '\0';
        ALOGI("Assertions enabled: '%s'\n", enableAssertBuf);
        opt.optionString = enableAssertBuf;
        mOptions.add(opt);
    } else {
        ALOGV("Assertions disabled\n");
    }

    if (jniOptsBuf[10] != '\0') {
        ALOGI("JNI options: '%s'\n", jniOptsBuf);
        opt.optionString = jniOptsBuf;
        mOptions.add(opt);
    }

    if (stackTraceFileBuf[0] != '\0') {
        static const char *stfOptName = "-Xstacktracefile:";

        stackTraceFile =
          (char*)malloc(strlen(stfOptName) + strlen(stackTraceFileBuf) +1);
        strcpy(stackTraceFile, stfOptName);
        strcat(stackTraceFile, stackTraceFileBuf);
        opt.optionString = stackTraceFile;
        mOptions.add(opt);
    }

    /* extra options; parse this late so it overrides others */
    property_get("dalvik.vm.extra-opts", extraOptsBuf, "");
    parseExtraOpts(extraOptsBuf);

    /* Set the properties for locale */
```

```
    {
        char langOption[sizeof("-Duser.language=") + 3];
        char regionOption[sizeof("-Duser.region=") + 3];
        strcpy(langOption, "-Duser.language=");
        strcpy(regionOption, "-Duser.region=");
        readLocale(langOption, regionOption);
        opt.extraInfo = NULL;
        opt.optionString = langOption;
        mOptions.add(opt);
        opt.optionString = regionOption;
        mOptions.add(opt);
    }

    /*
     * We don't have /tmp on the device, but we often have an SD card.  Apps
     * shouldn't use this, but some test suites might want to exercise it.
     */
    opt.optionString = "-Djava.io.tmpdir=/sdcard";
    mOptions.add(opt);

    initArgs.version = JNI_VERSION_1_4;
    initArgs.options = mOptions.editArray();
    initArgs.nOptions = mOptions.size();
    initArgs.ignoreUnrecognized = JNI_FALSE;

    /*
     * 初始化 VM.
     *
     * The JavaVM* is essentially per-process, and the JNIEnv* is per-thread.
     * If this call succeeds, the VM is ready, and we can start issuing
     * JNI calls.
     */
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }

    result = 0;

bail:
    free(stackTraceFile);
    return result;
}
```

在上述代码中，设置了一系列的控制 Dalvik VM 的选项，开发者可以通过特定的系统属性来指定这些选项。其中最为常用的选项有如下 4 个。

- -Xcheck:jni：功能是启动 JNI 检查方法。通过使用这个选项，可以检查要访问的 Java 对象的成员变量或者成员函数的合法性，例如检查类型是否匹配。在日常应用中，可

以使用系统属性 dalvik.vm.checkjni 或者 ro.kernel.android.checkjni 来指定是否要启用 -Xcheck:jni 选项。

- -Xint:portable/-Xint:fast/-Xint:jit：功能是指定 Dalvik 虚拟机的执行模式，可以通过系统属性 dalvik.vm.execution-mode 来指定 Dalvik VM 的解释模式。在 Dalvik VM 中支持如下三种运行模式。
  - Portable：表示 Dalvik VM 以可移植的方式来进行编译，编译出来的虚拟机可以在任意平台上运行。
  - Fast：可以针对当前平台的类型来编译 Dalvik VM。
  - Jit：将代码动态编译成本地语言后再执行。
- Xstacktracefile：功能是指定调用堆栈输出文件。当指定了-Xstacktracefile 选项之后，可以将线程的调用堆栈输出到指定的文件中去。
- -Xmx：功能是指定 Java 对象堆的最大值，Dalvik VM 的 Java 对象堆的默认最大值是 16MB。

5.4.3 创建并初始化Dalvik VM实例

调用函数 JNI_CreateJavaVM 创建并初始化一个 Dalvik VM 实例，此函数在文件 dalvik\vm\Jni.cpp 中定义，具体实现代码如下所示：

```
jint JNI_CreateJavaVM(JavaVM **p_vm, JNIEnv **p_env, void *vm_args) {
    const JavaVMInitArgs *args = (JavaVMInitArgs*)vm_args;
    if (args->version < JNI_VERSION_1_2) {
        return JNI_EVERSION;
    }

    // TODO: don't allow creation of multiple VMs -- one per customer for now

    /* zero globals; not strictly necessary the first time a VM is started */
    memset(&gDvm, 0, sizeof(gDvm));

    /*
     *为当前进程创建 Dalvik 虚拟机实例，即一个 JavaVMExt 对象
     */
    JavaVMExt *pVM = (JavaVMExt*)calloc(1, sizeof(JavaVMExt));
    pVM->funcTable = &gInvokeInterface;
    pVM->envList = NULL;
    dvmInitMutex(&pVM->envListLock);

    UniquePtr<const char*[]> argv(new const char*[args->nOptions]);
    memset(argv.get(), 0, sizeof(char*) * (args->nOptions));

    /*
     *为当前线程创建和初始化一个 JNI 环境
     *即一个 JNIEnvExt 对象
     *此功能是通过调用函数 dvmCreateJNIEnv 来完成的
```

```
 */
int argc = 0;
for (int i=0; i<args->nOptions; i++) {
    const char *optStr = args->options[i].optionString;
    if (optStr == NULL) {
        dvmFprintf(
          stderr, "ERROR: CreateJavaVM failed: argument %d was NULL\n", i);
        return JNI_ERR;
    } else if (strcmp(optStr, "vfprintf") == 0) {
        gDvm.vfprintfHook =
          (int (*)(FILE*, const char*, va_list))args->options[i].extraInfo;
    } else if (strcmp(optStr, "exit") == 0) {
        gDvm.exitHook = (void (*)(int)) args->options[i].extraInfo;
    } else if (strcmp(optStr, "abort") == 0) {
        gDvm.abortHook = (void (*)(void))args->options[i].extraInfo;
    } else if (strcmp(optStr, "sensitiveThread") == 0) {
        gDvm.isSensitiveThreadHook =
          (bool (*)(void))args->options[i].extraInfo;
    } else if (strcmp(optStr, "-Xcheck:jni") == 0) {
        gDvmJni.useCheckJni = true;
    } else if (strncmp(optStr, "-Xjniopts:", 10) == 0) {
        char *jniOpts = strdup(optStr + 10);
        size_t jniOptCount = 1;
        for (char *p=jniOpts; *p!=0; ++p) {
            if (*p == ',') {
                ++jniOptCount;
                *p = 0;
            }
        }
        char *jniOpt = jniOpts;
        for (size_t i=0; i<jniOptCount; ++i) {
            if (strcmp(jniOpt, "warnonly") == 0) {
                gDvmJni.warnOnly = true;
            } else if (strcmp(jniOpt, "forcecopy") == 0) {
                gDvmJni.forceCopy = true;
            } else if (strcmp(jniOpt, "logThirdPartyJni") == 0) {
                gDvmJni.logThirdPartyJni = true;
            } else {
                dvmFprintf(stderr,
                 "ERROR: CreateJavaVM failed: unknown -Xjniopts option '%s'\n",
                 jniOpt);
                return JNI_ERR;
            }
            jniOpt += strlen(jniOpt) + 1;
        }
        free(jniOpts);
    } else {
        /* regular option */
        argv[argc++] = optStr;
```

```
        }
    }

    if (gDvmJni.useCheckJni) {
        dvmUseCheckedJniVm(pVM);
    }

    if (gDvmJni.jniVm != NULL) {
        dvmFprintf(stderr, "ERROR: Dalvik only supports one VM per process\n");
        return JNI_ERR;
    }

    gDvmJni.jniVm = (JavaVM*)pVM;

    /*
     * Create a JNIEnv for the main thread.  We need to have something set up
     * here because some of the class initialization we do when starting
     * up the VM will call into native code.
     */
    JNIEnvExt *pEnv = (JNIEnvExt*)dvmCreateJNIEnv(NULL);

    /*将参数vm_args所描述的Dalvik VM启动选项拷贝到变量argv所描述的一个字符串数组中去。
    *调用函数dvmStartup来初始化前面所创建的Dalvik虚拟机实例
    */
    gDvm.initializing = true;
    std::string status =
       dvmStartup(argc, argv.get(), args->ignoreUnrecognized, (JNIEnv*)pEnv);
    gDvm.initializing = false;

    if (!status.empty()) {
        free(pEnv);
        free(pVM);
        ALOGW("CreateJavaVM failed: %s", status.c_str());
        return JNI_ERR;
    }

    /*
     * 调用函数dvmChangeStatus将当前线程的状态设置为正在执行NATIVE代码
     * 通过输出参数p_vm和p_env
     * 将刚刚创建和初始化好的JavaVMExt对象和JNIEnvExt对象返回给调用者
     */
    dvmChangeStatus(NULL, THREAD_NATIVE);
    *p_env = (JNIEnv*)pEnv;
    *p_vm = (JavaVM*)pVM;
    ALOGV("CreateJavaVM succeeded");
    return JNI_OK;
}
```

5.4.4　创建JNIEnvExt对象

再分析函数 dvmCreateJNIEnv，功能是创建 JNIEnvExt 对象以描述当前的 JNI 环境，并设置此 JNIEnvExt 对象的宿主 Dalvik VM 和所使用的本地接口表。此处的宿主 Dalvik VM 就是当前进程的 Dalvik VM，被保存在全局变量 gDvm 的成员变量 vmList 中。函数 dvmCreateJNIEnv 在文件 dalvik\vm\Jni.cpp 中定义，具体实现代码如下所示：

```
JNIEnv* dvmCreateJNIEnv(Thread *self) {
    JavaVMExt *vm = (JavaVMExt*)gDvmJni.jniVm;

    //if (self != NULL)
    //    ALOGI("Ent CreateJNIEnv: threadid=%d %p", self->threadId, self);

    assert(vm != NULL);

    JNIEnvExt *newEnv = (JNIEnvExt*)calloc(1, sizeof(JNIEnvExt));
    newEnv->funcTable = &gNativeInterface;
    if (self != NULL) {
        dvmSetJniEnvThreadId((JNIEnv*)newEnv, self);
        assert(newEnv->envThreadId != 0);
    } else {
        /* make it obvious if we fail to initialize these later */
        newEnv->envThreadId = 0x77777775;
        newEnv->self = (Thread*)0x77777779;
    }
    if (gDvmJni.useCheckJni) {
        dvmUseCheckedJniEnv(newEnv);
    }

    ScopedPthreadMutexLock lock(&vm->envListLock);

    /* insert at head of list */
    newEnv->next = vm->envList;
    assert(newEnv->prev == NULL);
    if (vm->envList == NULL) {
        // rare, but possible
        vm->envList = newEnv;
    } else {
        vm->envList->prev = newEnv;
    }
    vm->envList = newEnv;

    //if (self != NULL)
    //    ALOGI("Xit CreateJNIEnv: threadid=%d %p", self->threadId, self);
    return (JNIEnv*)newEnv;
}
```

在上述代码中，参数 self 表示前面创建的 JNIEnvExt 对象要关联的线程，通过源码分析可知，可以通过调用函数 dvmSetJniEnvThreadId 来将它们关联起来。

再看函数 dvmStartup，功能是初始化 Dalvik VM，并设置处理 Dalvik VM 的启动选项。这些启动选项都被保存在参数 argv 中，并且被保存选项的个数为 argc。在处理启动选项之前，会执行如下两个操作：

- 调用函数 setCommandLineDefaults 给 Dalvik VM 设置默认参数。
- 调用函数 processOptions 分配足够的内存空间，以容纳由参数 argv 和 argc 所描述的启动选项。

函数 dvmStartup 在文件 dalvik\vm\Init.cpp 中定义，具体实现代码如下所示：

```
std::string dvmStartup(int argc, const char* const argv[],
        bool ignoreUnrecognized, JNIEnv *pEnv)
{
    ScopedShutdown scopedShutdown;

    assert(gDvm.initializing);

    ALOGV("VM init args (%d):", argc);
    for (int i=0; i<argc; i++) {
        ALOGV("  %d: '%s'", i, argv[i]);
    }
    //调用函数 setCommandLineDefaults 来给 Dalvik 虚拟机设置默认参数
    //因为启动选项不一定会指定 Dalvik 虚拟机的所有属性
    setCommandLineDefaults();

    /*
     * 调用函数 processOptions 来分配足够的内存空间来容纳由参数 argv 和 argc 所描述的启动选项
     */
    int cc = processOptions(argc, argv, ignoreUnrecognized);
    if (cc != 0) {
        if (cc < 0) {
            dvmFprintf(stderr, "\n");
            usage("dalvikvm");
        }
        return "syntax error";
    }

#if WITH_EXTRA_GC_CHECKS > 1
    /* only "portable" interp has the extra goodies */
    if (gDvm.executionMode != kExecutionModeInterpPortable) {
        ALOGI("Switching to 'portable' interpreter for GC checks");
        gDvm.executionMode = kExecutionModeInterpPortable;
    }
#endif

    /* 配置调度选项 */
    if (!access("/dev/cpuctl/tasks", F_OK)) {
```

```
        ALOGV("Using kernel group scheduling");
        gDvm.kernelGroupScheduling = 1;
    } else {
        ALOGV("Using kernel scheduler policies");
    }

    /* 配置处理标志 */
    if (!gDvm.reduceSignals)
        blockSignals();

    /* 验证系统页面大小 */
    if (sysconf(_SC_PAGESIZE) != SYSTEM_PAGE_SIZE) {
        return StringPrintf("expected page size %d, got %d",
                SYSTEM_PAGE_SIZE, (int)sysconf(_SC_PAGESIZE));
    }

    /* mterp 设置 */
    ALOGV("Using executionMode %d", gDvm.executionMode);
    dvmCheckAsmConstants();

    /*
     *从下面的代码开始初始化 Dalvik 虚拟机的各个子模块
     */
    dvmQuasiAtomicsStartup();
    if (!dvmAllocTrackerStartup()) { //用于初始化 Dalvik 虚拟机的对象分配记录子模块
        return "dvmAllocTrackerStartup failed";
    }
    if (!dvmGcStartup()) { //用来初始化 Dalvik 虚拟机的垃圾收集(GC)子模块
        return "dvmGcStartup failed";
    }
    //dvmThreadStartup 用于初始化 Dalvik 虚拟机的线程列表
    //为主线程创建一个 Thread 对象以及为主线程初始化执行环境
    if (!dvmThreadStartup()) {
        return "dvmThreadStartup failed";
    }
    //用于初始化 Davlik 虚拟机的内建 Native 函数表
    if (!dvmInlineNativeStartup()) {
        return "dvmInlineNativeStartup";
    }
    //用于初始化寄存器映射集(Register Map)子模块
    if (!dvmRegisterMapStartup()) {
        return "dvmRegisterMapStartup failed";
    }
    //用于初始化 instanceof 操作符子模块
    if (!dvmInstanceofStartup()) {
        return "dvmInstanceofStartup failed";
    }
    //用于初始化启动类加载器(Bootstrap Class Loader)，同时初始化 java.lang.Class 类
    if (!dvmClassStartup()) {
```

```
        return "dvmClassStartup failed";
    }

    /*
     * At this point, the system is guaranteed to be sufficiently
     * initialized that we can look up classes and class members. This
     * call populates the gDvm instance with all the class and member
     * references that the VM wants to use directly.
     */
    if (!dvmFindRequiredClassesAndMembers()) {
        return "dvmFindRequiredClassesAndMembers failed";
    }
    //用于初始化java.lang.String类内部私有的一个字符串池，这样当Dalvik虚拟机运行起来之后
    //就可以调用java.lang.String类的成员函数intern来访问这个字符串池里面的字符串
    if (!dvmStringInternStartup()) {
        return "dvmStringInternStartup failed";
    }
    //用于初始化Native Shared Object库加载表，即SO库加载表
    if (!dvmNativeStartup()) {
        return "dvmNativeStartup failed";
    }
    //用于初始化内部Native函数表
    //所有需要直接访问Dalvik虚拟机内部函数或者数据结构的Native函数都定义在这张表中
    //因为它们如果定义在外部的其他SO文件中，就无法直接访问Dalvik虚拟机的内部函数或数据结构
    if (!dvmInternalNativeStartup()) {
        return "dvmInternalNativeStartup failed";
    }
    //用于初始化全局引用表，以及加载一些与Direct Buffer相关的类
    //例如DirectBuffer、PhantomReference和ReferenceQueue等
    if (!dvmJniStartup()) {
        return "dvmJniStartup failed";
    }
    //用于初始化Dalvik虚拟机的性能分析子模块，以及加载dalvik.system.VMDebug类等
    if (!dvmProfilingStartup()) {
        return "dvmProfilingStartup failed";
    }

    /*
     * Create a table of methods for which we will substitute an "inline"
     * version for performance.
     */
    if (!dvmCreateInlineSubsTable()) {
        return "dvmCreateInlineSubsTable failed";
    }

    /*
     *用于验证Dalvik虚拟机中存在相应的装箱类，并且这些装箱类有且仅有一个成员变量
     *这个成员变量是用来描述对应的数字值的。
     *这些装箱类包括 java.lang.Boolean、java.lang.Character、
```

```
 *java.lang.Float、java.lang.Double、java.lang.Byte、java.lang.Short、
 *java.lang.Integer 和 java.lang.Long
 */
if (!dvmValidateBoxClasses()) {
   return "dvmValidateBoxClasses failed";
}

/*
 *用于准备主线程的 JNI 环境。虽然我们已经为当前线程创建好一个 JNI 环境了，
 *但是还没有将该 JNI 环境与主线程关联，也就是还没有将主线程的 ID 设置到该 JNI 环境中去
 */
if (!dvmPrepMainForJni(pEnv)) {
   return "dvmPrepMainForJni failed";
}

/*
 * Explicitly initialize java.lang.Class.  This doesn't happen
 * automatically because it's allocated specially (it's an instance
 * of itself).  Must happen before registration of system natives,
 * which make some calls that throw assertions if the classes they
 * operate on aren't initialized.
 */
if (!dvmInitClass(gDvm.classJavaLangClass)) {
   return "couldn't initialized java.lang.Class";
}

/*
 *调用另外一个函数 jniRegisterSystemMethods，
 *后者接着又调用了函数 registerCoreLibrariesJni 来
 *为 Java 核心类注册 JNI 方法
 */
if (!registerSystemNatives(pEnv)) {
   return "couldn't register system natives";
}

/*
 *用于预创建一些与内存分配有关的异常对象，并且将它们缓存起来，以便以后可以快速使用。
 *这些异常对象包括 java.lang.OutOfMemoryError、java.lang.InternalError
 *和 java.lang.NoClassDefFoundError
 */
if (!dvmCreateStockExceptions()) {
   return "dvmCreateStockExceptions failed";
}

/*
 *用于为主线程创建一个 java.lang.ThreadGroup 对象、
 *java.lang.Thread 对象和 java.lang.VMThread 对象。
 *这些 Java 对象与在前面创建的 C++层的 Thread 对象关联一起，
 *共同用来描述 Dalvik 虚拟机的主线程
```

```
     */
    if (!dvmPrepMainThread()) {
        return "dvmPrepMainThread failed";
    }

    /*
     *用于确保主线程当前不引用有任何 Java 对象，是为了保证主线程接下来以干净的方式来执行程序入口
     */
    if (dvmReferenceTableEntries(&dvmThreadSelf()->internalLocalRefTable) != 0)
    {
        ALOGW("Warning: tracked references remain post-initialization");
        dvmDumpReferenceTable(&dvmThreadSelf()->internalLocalRefTable, "MAIN");
    }

    /*用于初始化 Dalvik 虚拟机的调试环境，
    *Dalvik VM 与 Java VM 一样，都是通过 JDWP 协议来支持远程调试的
    */
    if (!dvmDebuggerStartup()) {
        return "dvmDebuggerStartup failed";
    }

    if (!dvmGcStartupClasses()) {
        return "dvmGcStartupClasses failed";
    }

    /*
     * Init for either zygote mode or non-zygote mode.  The key difference
     * is that we don't start any additional threads in Zygote mode.
     */
    if (gDvm.zygote) {
        if (!initZygote()) {
            return "initZygote failed";
        }
    } else {
        if (!dvmInitAfterZygote()) {
            return "dvmInitAfterZygote failed";
        }
    }

#ifndef NDEBUG
    if (!dvmTestHash())
        ALOGE("dvmTestHash FAILED");
    if (false /*noisy!*/ && !dvmTestIndirectRefTable())
        ALOGE("dvmTestIndirectRefTable FAILED");
#endif

    if (dvmCheckException(dvmThreadSelf())) {
        dvmLogExceptionStackTrace();
        return "Exception pending at end of VM initialization";
```

```
    }

    scopedShutdown.disarm();
    return "";
}
```

5.4.5　设置当前进程

再看函数 dvmInitZygote，功能是调用系统中的 setpgid 函数来设置当前进程，设置 Zyogte 进程的进程组 ID。当系统在调用 setpgid 时会传递两个都为 0 的参数，这表示 Zyogte 进程的进程组 ID 与进程 ID 相同，即 Zyogte 进程运行在一个单独的进程组里面。函数 dvmInitZygote 在文件 dalvik/vm/Init.c 中定义，具体实现代码如下所示：

```
static bool dvmInitZygote(void)
{
    /* zygote goes into its own process group */
    setpgid(0,0);

    return true;
}
```

5.4.6　注册Android核心类的JNI方法

接下来看函数 startReg，在文件 AndroidRuntime.java 中，函数 start 调用了类 AndroidRuntime 类中的成员函数 startReg 来注册 Android 核心类的 JNI 方法。函数 startReg 在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，具体实现代码如下所示：

```
/*
 * Register android native functions with the VM.
 */
/*static*/
int AndroidRuntime::startReg(JNIEnv *env)
{
    /*
     *调用函数 androidSetCreateThreadFunc 设置一个线程创建钩子 javaCreateThreadEtc。
     *此线程创建钩子是用来初始化一个 Native 线程的 JNI 环境的，
     *即当我们在 C++代码中创建一个 Native 线程的时候，
     *函数 javaCreateThreadEtc 会被调用来初始化该 Native 线程的 JNI 环境
     */
    androidSetCreateThreadFunc((android_create_thread_fn)javaCreateThreadEtc);

    LOGV("--- registering native functions ---\n");

    env->PushLocalFrame(200);
    /*
     *调用函数 register_jni_procs 来注册 Android 核心类的 JNI 方法。在注册 JNI 方法的过程中，
     *需要在 Native 代码中引用到一些 Java 对象，
```

```
     *这些 Java 对象引用需要记录在当前线程的一个 Native 堆栈中。
     *但是此时 Dalvik 虚拟机还没有真正运行起来，也就是当前线程的 Native 堆栈还没有准备就绪。
     *此时需要在注册 JNI 方法之前，手动地将在当前线程的 Native 堆栈中压入一个帧(Frame)，
     *并且在注册 JNI 方法之后，手动地将该帧弹出来
     */
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);
    //createJavaThread("fubar", quickTest, (void*)"hello");
    return 0;
}
```

在上述代码中，参数 env 指向了一个 JNIEnv 对象，此对象表示当前线程的 JNI 环境。通过调用 JNIEnv 对象的成员函数 PushLocalFrame 和 PopLocalFrame，可以用手动的方式向当前线程的 Native 堆栈中分别压入和弹出一个帧。这里的帧是一个本地帧，只能保存 Java 对象在 Native(本地)代码中的本地引用。函数 register_jni_procs 的具体实现代码如下所示：

```
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv *env)
{
    for (size_t i=0; i<count; i++) {
        if (array[i].mProc(env) < 0) {
#ifndef NDEBUG
            ALOGD("----------!!! %s failed to load\n", array[i].mName);
#endif
            return -1;
        }
    }
    return 0;
}
```

在上述代码中，参数 array 指向了全局变量 gRegJNI 所描述的 JNI 方法注册函数表，每一个表的选项都用一个 RegJNIRec 对象来描述，而每一个 RegJNIRec 对象都有一个成员变量 mProc 指向一个 JNI 方法注册函数，通过依次调用这些注册函数的方式，即可将 JNI 方法注册到创建的 Dalvik VM 中去。

在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中一一列出了全局变量 gRegJNI 所描述的 JNI 方法注册函数表，在此可以了解注册了哪些 Android 核心类的 JNI 方法。对应的代码如下所示：

```
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    REG_JNI(register_android_util_FloatMath),
    REG_JNI(register_android_text_format_Time),
```

```
    REG_JNI(register_android_content_AssetManager),
    REG_JNI(register_android_content_StringBlock),
    REG_JNI(register_android_content_XmlBlock),
    REG_JNI(register_android_emoji_EmojiFactory),
    REG_JNI(register_android_text_AndroidCharacter),
    REG_JNI(register_android_text_AndroidBidi),
    REG_JNI(register_android_view_InputDevice),
    REG_JNI(register_android_view_KeyCharacterMap),
    REG_JNI(register_android_os_Process),
    REG_JNI(register_android_os_SystemProperties),
    REG_JNI(register_android_os_Binder),
    REG_JNI(register_android_os_Parcel),
    REG_JNI(register_android_view_DisplayEventReceiver),
    REG_JNI(register_android_nio_utils),
    REG_JNI(register_android_graphics_PixelFormat),
    REG_JNI(register_android_graphics_Graphics),
    REG_JNI(register_android_view_GLES20DisplayList),
    REG_JNI(register_android_view_GLES20Canvas),
    REG_JNI(register_android_view_HardwareRenderer),
    REG_JNI(register_android_view_Surface),
    REG_JNI(register_android_view_SurfaceControl),
    REG_JNI(register_android_view_SurfaceSession),
    REG_JNI(register_android_view_TextureView),
    REG_JNI(register_com_google_android_gles_jni_EGLImpl),
    REG_JNI(register_com_google_android_gles_jni_GLImpl),
    REG_JNI(register_android_opengl_jni_EGL14),
    REG_JNI(register_android_opengl_jni_EGLExt),
    REG_JNI(register_android_opengl_jni_GLES10),
    REG_JNI(register_android_opengl_jni_GLES10Ext),
    REG_JNI(register_android_opengl_jni_GLES11),
    REG_JNI(register_android_opengl_jni_GLES11Ext),
    REG_JNI(register_android_opengl_jni_GLES20),
    REG_JNI(register_android_opengl_jni_GLES30),

    REG_JNI(register_android_graphics_Bitmap),
    REG_JNI(register_android_graphics_BitmapFactory),
    REG_JNI(register_android_graphics_BitmapRegionDecoder),
    REG_JNI(register_android_graphics_Camera),
    REG_JNI(register_android_graphics_Canvas),
    REG_JNI(register_android_graphics_ColorFilter),
    REG_JNI(register_android_graphics_DrawFilter),
    REG_JNI(register_android_graphics_Interpolator),
    REG_JNI(register_android_graphics_LayerRasterizer),
    REG_JNI(register_android_graphics_MaskFilter),
    REG_JNI(register_android_graphics_Matrix),
    REG_JNI(register_android_graphics_Movie),
    REG_JNI(register_android_graphics_NinePatch),
    REG_JNI(register_android_graphics_Paint),
    REG_JNI(register_android_graphics_Path),
```

```
    REG_JNI(register_android_graphics_PathMeasure),
    REG_JNI(register_android_graphics_PathEffect),
    REG_JNI(register_android_graphics_Picture),
    REG_JNI(register_android_graphics_PorterDuff),
    REG_JNI(register_android_graphics_Rasterizer),
    REG_JNI(register_android_graphics_Region),
    REG_JNI(register_android_graphics_Shader),
    REG_JNI(register_android_graphics_SurfaceTexture),
    REG_JNI(register_android_graphics_Typeface),
    REG_JNI(register_android_graphics_Xfermode),
    REG_JNI(register_android_graphics_YuvImage),

    REG_JNI(register_android_database_CursorWindow),
    REG_JNI(register_android_database_SQLiteConnection),
    REG_JNI(register_android_database_SQLiteGlobal),
    REG_JNI(register_android_database_SQLiteDebug),
    REG_JNI(register_android_os_Debug),
    REG_JNI(register_android_os_FileObserver),
    REG_JNI(register_android_os_FileUtils),
    REG_JNI(register_android_os_MessageQueue),
    REG_JNI(register_android_os_ParcelFileDescriptor),
    REG_JNI(register_android_os_SELinux),
    REG_JNI(register_android_os_Trace),
    REG_JNI(register_android_os_UEventObserver),
    REG_JNI(register_android_net_LocalSocketImpl),
    REG_JNI(register_android_net_NetworkUtils),
    REG_JNI(register_android_net_TrafficStats),
    REG_JNI(register_android_net_wifi_WifiManager),
    REG_JNI(register_android_os_MemoryFile),
    REG_JNI(register_com_android_internal_os_ZygoteInit),
    REG_JNI(register_android_hardware_Camera),
    REG_JNI(register_android_hardware_SensorManager),
    REG_JNI(register_android_hardware_SerialPort),
    REG_JNI(register_android_hardware_UsbDevice),
    REG_JNI(register_android_hardware_UsbDeviceConnection),
    REG_JNI(register_android_hardware_UsbRequest),
    REG_JNI(register_android_media_AudioRecord),
    REG_JNI(register_android_media_AudioSystem),
    REG_JNI(register_android_media_AudioTrack),
    REG_JNI(register_android_media_JetPlayer),
    REG_JNI(register_android_media_RemoteDisplay),
    REG_JNI(register_android_media_ToneGenerator),

    REG_JNI(register_android_opengl_classes),
    REG_JNI(register_android_server_NetworkManagementSocketTagger),
    REG_JNI(register_android_server_Watchdog),
    REG_JNI(register_android_ddm_DdmHandleNativeHeap),
    REG_JNI(register_android_backup_BackupDataInput),
    REG_JNI(register_android_backup_BackupDataOutput),
```

```
    REG_JNI(register_android_backup_FileBackupHelperBase),
    REG_JNI(register_android_backup_BackupHelperDispatcher),
    REG_JNI(register_android_app_backup_FullBackup),
    REG_JNI(register_android_app_ActivityThread),
    REG_JNI(register_android_app_NativeActivity),
    REG_JNI(register_android_view_InputChannel),
    REG_JNI(register_android_view_InputEventReceiver),
    REG_JNI(register_android_view_InputEventSender),
    REG_JNI(register_android_view_InputQueue),
    REG_JNI(register_android_view_KeyEvent),
    REG_JNI(register_android_view_MotionEvent),
    REG_JNI(register_android_view_PointerIcon),
    REG_JNI(register_android_view_VelocityTracker),

    REG_JNI(register_android_content_res_ObbScanner),
    REG_JNI(register_android_content_res_Configuration),

    REG_JNI(register_android_animation_PropertyValuesHolder),
    REG_JNI(register_com_android_internal_content_NativeLibraryHelper),
    REG_JNI(register_com_android_internal_net_NetworkStatsFactory),
};
```

5.4.7　使用线程创建javaCreateThreadEtc钩子

在文件 AndroidRuntime.java 中，函数 start 调用了 AndroidRuntime 类中的成员函数 androidSetCreateThreadFunc，这样可以使用线程创建 javaCreateThreadEtc 钩子。

函数 androidSetCreateThreadFunc 在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，具体实现代码如下所示：

```
void androidSetCreateThreadFunc(android_create_thread_fn func)
{
    gCreateThreadFn = func;
}
```

由上述实现代码可知，函数指针 gCreateThreadFn 中保存了线程创建的 javaCreateThreadEtc 钩子。函数指针 gCreateThreadFn 默认时指向函数 androidCreateRawThreadEtc，如果不设置线程创建钩子，则函数 androidCreateRawThreadEtc 就是默认使用的线程创建函数。

5.5　创建Dalvik VM进程

在 Android 系统中，Dalvik VM 不但可以执行 Java 代码，而且还可以执行 Native(本地)代码，也就是 C/C++函数。在执行这些 C/C++函数的过程中，可以通过本地操作系统提供的系统调用创建 Linux 进程和线程。如果在 Native 代码中创建出来的线程能够执行 Java 代码，那么它实际上又可以看作是一个 Dalvik VM 线程。

在 Android 系统中，通过类 android.os.Process 的静态成员函数 start 来创建 Dalvik VM 进程。即由 ActivityManagerService 服务通过类 android.os.Process 的静态成员函数 start 来请求 Zygote 进程的方式创建，而 Zyogte 进程又是通过类 dalvik.system.Zygote 的静态成员函数 forkAndSpecialize 创建该 Android 应用程序进程的。

5.5.1 分析底层启动过程

首先看函数 forkAndSpecialize，这是一个 JNI 函数，此函数在文件 libcore/dalvik/src/main/java/dalvik/system/Zygote.java 中定义，具体实现代码如下所示：

```
native public static int forkAndSpecialize(int uid, int gid, int[] gids,
 int debugFlags, int[][] rlimits);
   ...
}
```

函数 forkAndSpecialize 是由 C++层的函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 来实现的，此函数在文件 dalvik/vm/native/dalvik_system_Zygote.cpp 中定义，具体实现代码如下所示：

```
static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4 *args,
 JValue *pResult)
{
   pid_t pid;
   pid = forkAndSpecializeCommon(args, false);
   RETURN_INT(pid);
}
```

各个参数的具体说明如下所示。

- args：指向了一个 u4 数组，在里面包含了由 Dalvik 虚拟机封装的所有从 Java 层传递进来的参数。
- pResult：用于保存 JNI 方法调用的结果，这是通过宏 RETURN_INT 来实现的。

5.5.2 创建Dalvik VM进程

函数 Dalvik_dalvik_system_Zygote_forkAndSpec 中调用了函数 forkAndSpecializeCommon，功能是创建一个 Dalvik VM 进程。

函数 forkAndSpecializeCommon 在文件 dalvik/vm/native/dalvik_system_Zygote.cpp 中定义，具体实现代码如下所示：

```
static pid_t forkAndSpecializeCommon(const u4 *args, bool isSystemServer)
{
   pid t pid;

   uid_t uid = (uid_t)args[0];
   gid t gid = (gid t)args[1];
   ArrayObject *gids = (ArrayObject*)args[2];
   u4 debugFlags = args[3];
```

```
ArrayObject *rlimits = (ArrayObject*)args[4];
u4 mountMode = MOUNT_EXTERNAL_NONE;
int64_t permittedCapabilities, effectiveCapabilities;
char *seInfo = NULL;
char *niceName = NULL;

if (isSystemServer) {
    /*
     * Don't use GET_ARG_LONG here for now.  gcc is generating code
     * that uses register d8 as a temporary, and that's coming out
     * scrambled in the child process.  b/3138621
     */
    //permittedCapabilities = GET_ARG_LONG(args, 5);
    //effectiveCapabilities = GET_ARG_LONG(args, 7);
    permittedCapabilities = args[5] | (int64_t) args[6] << 32;
    effectiveCapabilities = args[7] | (int64_t) args[8] << 32;
} else {
    mountMode = args[5];
    permittedCapabilities = effectiveCapabilities = 0;
    StringObject *seInfoObj = (StringObject*)args[6];
    if (seInfoObj) {
        seInfo = dvmCreateCstrFromString(seInfoObj);
        if (!seInfo) {
            ALOGE("seInfo dvmCreateCstrFromString failed");
            dvmAbort();
        }
    }
    StringObject *niceNameObj = (StringObject*)args[7];
    if (niceNameObj) {
        niceName = dvmCreateCstrFromString(niceNameObj);
        if (!niceName) {
            ALOGE("niceName dvmCreateCstrFromString failed");
            dvmAbort();
        }
    }
}

if (!gDvm.zygote) {
    dvmThrowIllegalStateException(
        "VM instance not started with -Xzygote");

    return -1;
}

if (!dvmGcPreZygoteFork()) {
    ALOGE("pre-fork heap failed");
    dvmAbort();
}
```

```
    setSignalHandler();

    dvmDumpLoaderStats("zygote");
    pid = fork();

    if (pid == 0) {
        int err;
        /* The child process */

#ifdef HAVE_ANDROID_OS
        extern int gMallocLeakZygoteChild;
        gMallocLeakZygoteChild = 1;

        /* keep caps across UID change, unless we're staying root */
        if (uid != 0) {
            err = prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0);

            if (err < 0) {
                ALOGE("cannot PR_SET_KEEPCAPS: %s", strerror(errno));
                dvmAbort();
            }
        }

        for (int i=0; prctl(PR_CAPBSET_READ, i, 0, 0, 0)>=0; i++) {
            err = prctl(PR_CAPBSET_DROP, i, 0, 0, 0);
            if (err < 0) {
                if (errno == EINVAL) {
                    ALOGW("PR_CAPBSET_DROP %d failed: %s. "
                          "Please make sure your kernel is compiled with "
                          "file capabilities support enabled.",
                          i, strerror(errno));
                } else {
                    ALOGE("PR_CAPBSET_DROP %d failed: %s.", i, strerror(errno));
                    dvmAbort();
                }
            }
        }

#endif /* HAVE_ANDROID_OS */

        if (mountMode != MOUNT_EXTERNAL_NONE) {
            err = mountEmulatedStorage(uid, mountMode);
            if (err < 0) {
                ALOGE("cannot mountExternalStorage(): %s", strerror(errno));

                if (errno == ENOTCONN || errno == EROFS) {
                    // When device is actively encrypting, we get ENOTCONN here
                    // since FUSE was mounted before the framework restarted.
                    // When encrypted device is booting, we get EROFS since
```

```
            // FUSE hasn't been created yet by init.
            // In either case, continue without external storage.
        } else {
            dvmAbort();
        }
    }
}

err = setgroupsIntarray(gids);
if (err < 0) {
    ALOGE("cannot setgroups(): %s", strerror(errno));
    dvmAbort();
}

err = setrlimitsFromArray(rlimits);
if (err < 0) {
    ALOGE("cannot setrlimit(): %s", strerror(errno));
    dvmAbort();
}

err = setresgid(gid, gid, gid);
if (err < 0) {
    ALOGE("cannot setresgid(%d): %s", gid, strerror(errno));
    dvmAbort();
}

err = setresuid(uid, uid, uid);
if (err < 0) {
    ALOGE("cannot setresuid(%d): %s", uid, strerror(errno));
    dvmAbort();
}

if (needsNoRandomizeWorkaround()) {
    int current = personality(0xffffFFFF);
    int success = personality((ADDR_NO_RANDOMIZE | current));
    if (success == -1) {
        ALOGW("Personality switch failed. current=%d error=%d\n",
              current, errno);
    }
}

err = setCapabilities(permittedCapabilities, effectiveCapabilities);
if (err != 0) {
    ALOGE("cannot set capabilities (%llx,%llx): %s",
        permittedCapabilities, effectiveCapabilities, strerror(err));
    dvmAbort();
}

err = set_sched_policy(0, SP_DEFAULT);
```

```
        if (err < 0) {
            ALOGE("cannot set sched policy(0, SP DEFAULT): %s", strerror(-err));
            dvmAbort();
        }

        err = setSELinuxContext(uid, isSystemServer, seInfo, niceName);
        if (err < 0) {
            ALOGE("cannot set SELinux context: %s\n", strerror(errno));
            dvmAbort();
        }
        // These free(3) calls are safe because we know we're only ever forking
        // a single-threaded process, so we know no other thread held the heap
        // lock when we forked.
        free(seInfo);
        free(niceName);

        /*
         * Our system thread ID has changed.  Get the new one.
         */
        Thread *thread = dvmThreadSelf();
        thread->systemTid = dvmGetSysThreadId();

        /* configure additional debug options */
        enableDebugFeatures(debugFlags);

        unsetSignalHandler();
        gDvm.zygote = false;
        if (!dvmInitAfterZygote()) {
            ALOGE("error in post-zygote initialization");
            dvmAbort();
        }
    } else if (pid > 0) {
        /* the parent process */
        free(seInfo);
        free(niceName);
    }

    return pid;
}
```

当函数 forkAndSpecializeCommon 创建 System 进程时，参数 isSystemServer 的值等于 true，此时在参数列表 args 会包含两个额外的参数 permittedCapabilities 和 effectiveCapabilities。其中前者表示 System 进程允许的特权，而后者表示 System 进程当前的有效特权。

5.5.3 初始化运行的Dalvik VM

另外，在函数 forkAndSpecializeCommon 中还调用了函数 dvmInitAfterZygote，功能是进一步初始化在新创建的进程中运行的 Dalvik VM。函数 dvmInitAfterZygote 在文件 dalvik/vm/

Init.cpp 中定义，具体实现代码如下所示：

```
bool dvmInitAfterZygote()
{
    u8 startHeap, startQuit, startJdwp;
    u8 endHeap, endQuit, endJdwp;

    startHeap = dvmGetRelativeTimeUsec();

    /*
     * Post-zygote heap initialization, including starting
     * the HeapWorker thread.
     */
    if (!dvmGcStartupAfterZygote())
        return false;

    endHeap = dvmGetRelativeTimeUsec();
    startQuit = dvmGetRelativeTimeUsec();

    /* start signal catcher thread that dumps stacks on SIGQUIT */
    if (!gDvm.reduceSignals && !gDvm.noQuitHandler) {
        if (!dvmSignalCatcherStartup())
            return false;
    }

    /* start stdout/stderr copier, if requested */
    if (gDvm.logStdio) {
        if (!dvmStdioConverterStartup())
            return false;
    }

    endQuit = dvmGetRelativeTimeUsec();
    startJdwp = dvmGetRelativeTimeUsec();

    /*
     * Start JDWP thread.  If the command-line debugger flags specified
     * "suspend=y", this will pause the VM.  We probably want this to
     * come last.
     */
    if (!initJdwp()) {
        ALOGD("JDWP init failed; continuing anyway");
    }

    endJdwp = dvmGetRelativeTimeUsec();

    ALOGV("thread-start heap=%d quit=%d jdwp=%d total=%d usec",
         (int)(endHeap-startHeap), (int)(endQuit-startQuit),
         (int)(endJdwp-startJdwp), (int)(endJdwp-startHeap));
```

```
#ifdef WITH_JIT
    if (gDvm.executionMode == kExecutionModeJit) {
        if (!dvmCompilerStartup())
            return false;
    }
#endif

    return true;
}
```

到此为止，一个 Dalvik VM 进程的创建工作就完成了。由此可以看出：Dalvik VM 进程的实质就是 Linux 进程。

第 6 章 IPC通信机制详解

在 Android 系统中，应用程序都是由 Activity 和 Service 组成的，其中 Service 通常运行在独立的进程中，而 Activity 不但可以运行在同一个进程中，而且也可以运行在不同的进程中。

那么，不在同一个进程中的 Activity 或 Service 是如何实现通信功能的呢？是通过 Android 系统的 Binder 进程通信机制实现的。

在本章的内容中，将详细分析 Android 4.3 系统中的 IPC 进程通信机制的实现源码，为读者步入本书后面知识的学习打下基础。

6.1 Binder机制概述

关于 Binder，我们曾在第 4 章做过一些介绍，这里回顾后将深入探讨。

Binder 是 Android 系统提供的一种 IPC(进程间通信)机制。由于 Android 是基于 Linux 内核的，因此，除了 Binder 以外，还存在其他的 IPC 机制，例如管道和 socket 等。Binder 相对于其他 IPC 机制来说，就更加灵活和方便了。

Binder 的驱动代码在 kernel/drivers/staing/android/binder.c 中，另外，该目录下还有一个 binder.h 头文件。Binder 是一个虚拟设备，所以它的代码相对而言还算简单，读者只要有基本的 Linux 驱动开发方面的知识就能读懂它。/proc/binder 目录下的内容可用来查看 Binder 设备的运行状况。

对于初学 Android 的读者来说，最难掌握的可能就是 Binder 机制了，因为 Android 系统基本上可以看作是一个基于 Binder 通信的 C/S 架构。Binder 就像网络一样，把系统的各个部分连接在了一起，因此它是非常重要的。在基于 Binder 通信的 C/S 架构体系中，除了 C/S 架构所包括的 Client 端和 Server 端外，Android 还有一个全局的 ServiceManager 端，它的作用是管理系统中的各种服务(Service)。

Android 系统的 Binder 机制由 Client、Server、Service Manager 组成，如图 6-1 所示。

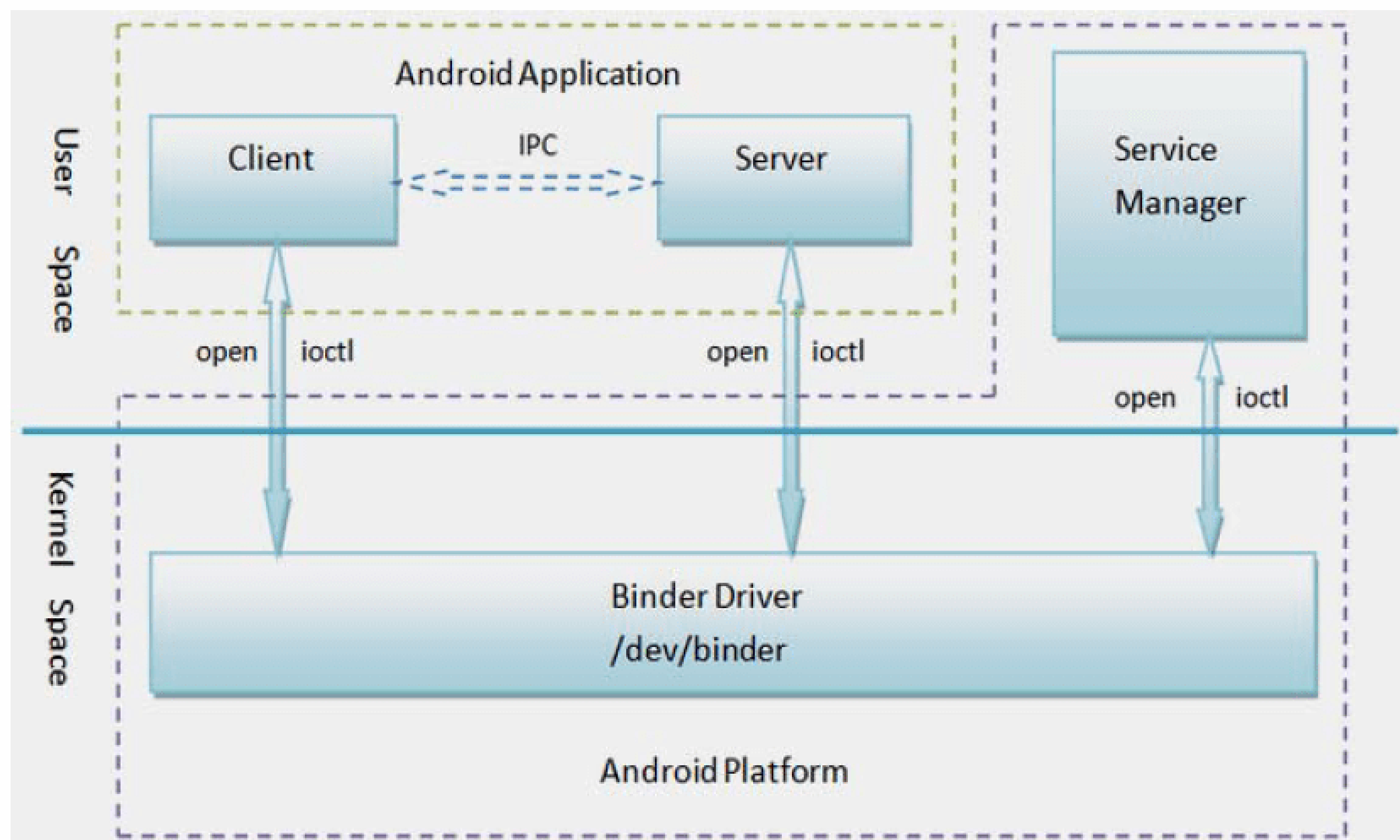


图 6-1 Binder机制中的组件关系

在 Android 系统中，Client、Server 和 ServiceManager 之间的交互关系如下所示：

- Client、Server 和 Service Manager 在用户空间中实现，Binder 驱动程序在内核空间中实现。
- Server 进程要先注册一些 Service 到 ServiceManager 中，所以 Server 是 ServiceManager 的客户端，而 ServiceManager 就是服务端了。ServiceManager 是一个守护进程，能够管理 Server 并向 Client 提供查询 Server 的接口。
- 如果某个 Client 进程要使用某个 Service，必须先到 ServiceManager 中获取该 Service

的相关信息，所以 Client 是 ServiceManager 的客户端。另外，Client 根据得到的 Service 信息与 Service 所在的 Server 进程建立通信的通路，然后就可以直接与 Service 交互了，所以 Client 也是 Server 的客户端。

- Binder 驱动程序提供设备文件/dev/binder 与用户空间交互，Client、Server 和 Service Manager 通过 open 和 ioctl 文件操作函数与 Binder 驱动程序进行通信。
- 在 Android 平台中，已经实现了 Binder 驱动程序和 Service Manager，开发者只需要在用户空间实现自己的 Client 和 Server 即可。
- 三者的交互都是基于 Binder 通信的，所以通过任意两者之间的关系，都可以揭示 Binder 的奥秘。

6.2　分析Binder驱动程序

Binder 采用 AIDL(Android Interface Description Language)来描述进程间通信的接口。Binder 作为一个特殊的字符设备，其设备节点是/dev/binder。主要代码在如下文件中实现：

```
kernel/drivers/staging/binder.h
kernel/drivers/staging/binder.c
```

在本节的内容中，将详细分析上述文件的实现源码。

6.2.1　分析数据结构

在 Binder 驱动程序中，主要包含了如下所示的数据结构。

(1)　binder_work

binder_work 表示在 Binder 驱动中进程所要处理的工作项，定义代码如下所示：

```
struct binder_work {
    struct list_head entry;
    enum {
        BINDER_WORK_TRANSACTION = 1,
        BINDER_WORK_TRANSACTION_COMPLETE,
        BINDER_WORK_NODE,
        BINDER_WORK_DEAD_BINDER,
        BINDER_WORK_DEAD_BINDER_AND_CLEAR,
        BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
    } type;
};
```

在上述结构体定义中，entry 被定义为 list_head 类型，用于实现一个双向链表，能够存储所有 binder_work 的队列；并且还包含了一个 enum 类型的 type：binder_work。

(2)　binder_node

结构体 binder_node 用来定义 Binder 实体对象。在 Android 系统中，每一个 Service 组件在 Binder 驱动程序中都有一个 Binder 实体对象。

定义 binder_node 的代码如下所示：

```
struct binder_node {
    int debug_id;
    struct binder_work work;
    union {
        struct rb_node rb_node;
        struct hlist_node dead_node;
    };
    struct binder_proc *proc;
    struct hlist_head refs;
    int internal_strong_refs;
    int local_weak_refs;
    int local_strong_refs;
    void __user *ptr;
    void __user *cookie;
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    unsigned has_async_transaction:1;
    unsigned accept_fds:1;
    unsigned min_priority:8;
    struct list_head async_todo;
};
```

驱动中的 Binder 实体也叫作“节点”，隶属于提供实体的进程。结构体 binder_node 中，各个成员的具体说明如表 6-1 所示。

表 6-1　结构体binder_node中的成员说明

成　员	含　义
int debug_id;	用于调试
struct binder_work work;	当本节点引用计数发生改变，需要通知所属进程时，通过该成员挂入所属进程的 to-do 队列里，唤醒所属进程执行 Binder 实体引用计数的修改

union {
struct rb_node rb_node;
struct hlist_node dead_node;
};
	每个进程都维护一棵红黑树，以 Binder 实体在用户空间的指针，即本结构的 ptr 成员为索引存放该进程所有的 Binder 实体。这样驱动可以根据 Binder 实体在用户空间的指针很快找到其位于内核的节点。rb_node 用于将本节点链入该红黑树中。
	销毁节点时，须将 rb_node 从红黑树中摘除，但如果本节点还有引用没有切断，就用 dead_node 将节点隔离到另一个链表中，直到通知所有进程切断与该节点的引用后，该节点才可能被销毁

struct binder_proc *proc;	本成员指向节点所属的进程，即提供该节点的进程
struct hlist_head refs;	本成员是队列头，所有指向本节点的引用都链接在该队列里。这些引用可能隶属于不同的进程。通过该队列可以遍历指向该节点的所有引用
int internal_strong_refs;	用以实现强指针的计数器：产生一个指向本节点的强引用时该计数就会加 1

续表

成　员	含　义
int local_weak_refs;	驱动为传输中的 Binder 设置的弱引用计数。如果一个 Binder 打包在数据包中，从一个进程发送到另一个进程，驱动会为该 Binder 增加引用计数，直到接收进程通过 BC_FREE_BUFFER 通知驱动释放该数据包的数据区为止
int local_strong_refs;	驱动为传输中的 Binder 设置的强引用计数。同上
void __user *ptr;	指向用户空间 Binder 实体的指针，来自于 flat_binder_object 的 binder 成员
void __user *cookie;	指向用户空间的附加指针，来自于 flat_binder_object 的 cookie 成员
unsigned has_strong_ref; unsigned pending_strong_ref; unsigned has_weak_ref; unsigned pending_weak_ref	这一组标志用于控制驱动与 Binder 实体所在进程交互式修改引用计数
unsigned has_async_transaction;	该成员表明该节点在 to-do 队列中有异步交互尚未完成。驱动将所有发送往接收端的数据包暂存在接收进程或线程开辟的 to-do 队列里。对于异步交互，驱动做了适当流控：如果 to-do 队列里有异步交互尚待处理，则该成员置 1，这将导致新到的异步交互存放在本结构成员的– asynch_todo 队列中，而不直接送到 to-do 队列里，目的是为同步交互让路，避免长时间阻塞发送端
unsigned accept_fds	表明节点是否同意接受文件方式的 Binder，来自 flat_binder_object 中 flags 成员的 FLAT_BINDER_FLAG_ACCEPTS_FDS 位。由于接收文件 Binder 会为进程自动打开一个文件，占用有限的文件描述符，节点可以设置该位拒绝这种行为
int min_priority	设置处理 Binder 请求的线程的最低优先级。发送线程将数据提交给接收线程处理时，驱动会将发送线程的优先级也赋予接收线程，使得数据即使跨了进程，也能以同样优先级得到处理。不过如果发送线程优先级过低，接收线程将以预设的最小值运行。 该域的值来自于 flat_binder_object 中 flags 成员
struct list_head async_todo	异步交互等待队列；用于分流发往本节点的异步交互包

(3)　binder_ref

结构体 binder_ref 用来描述一个 Binder 引用对象，在 Android 系统中，每一个 Client 组件在 Binder 驱动程序中都有一个 Binder 引用对象。定义 binder_ref 的代码如下所示：

```
struct binder_ref {
    /* Lookups needed: */
    /*   node + proc => ref (transaction) */
    /*   desc + proc => ref (transaction, inc/dec ref) */
    /*   node => refs + procs (proc exit) */
    int debug_id;
    struct rb_node rb_node_desc;
    struct rb_node rb_node_node;
```

```
    struct hlist_node node_entry;
    struct binder_proc *proc;
    struct binder_node *node;
    uint32_t desc;
    int strong;
    int weak;
    struct binder_ref_death *death;
};
```

结构体 binder_ref 中，各个成员的具体说明如表 6-2 所示。

表 6-2　结构体binder_ref中的成员说明

成　员	含　义
int debug_id;	调试用
struct rb_node rb_node_desc;	每个进程有一棵红黑树，进程所有引用以引用号(即本结构的 desc 域)为索引添入该树中。本成员用做链接到该树的一个节点
struct rb_node rb_node_node;	每个进程又有一棵红黑树，进程所有引用以节点实体在驱动中的内存地址(即本结构的 node 域)为索引添入该树中。本成员用作链接到该树的一个节点
struct hlist_node node_entry;	该域将本引用作为节点链入所指向的 Binder 实体结构 binder_node 中的 refs 队列
struct binder_proc *proc;	本引用所属的进程
struct binder_node *node;	本引用所指向的节点(Binder 实体)
uint32_t desc;	本结构的引用号
int strong;	强引用计数
int weak;	弱引用计数
struct binder_ref_death *death;	应用程序向驱动发送 BC_REQUEST_DEATH_NOTIFICATION 或 BC_CLEAR_DEATH_NOTIFICATION 命令，从而当 Binder 实体销毁时，能够收到来自驱动的提醒。该域不为空表明用户订阅了对应实体销毁的“噩耗”

(4)　binder_ref_death

binder_ref_death 是一个通知结构体，只要某进程对某 Binder 引用订阅了其实体的死亡通知，那么 Binder 驱动将会为该 Binder 引用建立一个 binder_ref_death 通知结构体，将其保存在当前进程的对应 Binder 引用结构体的 death 域中。定义 binder_ref_death 的代码如下所示：

```
struct binder_ref_death {
    struct binder_work work;
    void __user *cookie;
};
```

(5)　binder_buffer

结构体 binder_buffer 用来描述一个内核缓冲区，能够在进程之间传输数据。

定义 binder_buffer 的代码如下所示：

```
struct binder_buffer {
    struct list_head entry; /* free and allocated entries by address */
    struct rb_node rb_node; /* free entry by size or allocated entry */
                            /* by address */
    unsigned free:1;
    unsigned allow_user_free:1;
    unsigned async_transaction:1;
    unsigned debug_id:29;

    struct binder_transaction *transaction;

    struct binder_node *target_node;
    size_t data_size;
    size_t offsets_size;
    uint8_t data[0];
};
```

结构体 binder_buffer 能够储存 Binder 的相关信息，成员的具体说明如下所示。

- entry：构建一个双向链表。
- rb_node：表示一个红黑树节点。
- transaction：用于中转请求和返回结果。
- target_node：是一个目标节点。
- data_size：表示数据的大小。
- offsets_size：是一个偏移量。
- data[0]：用于存储实际数据。

(6)　binder_proc

结构体 binder_proc 表示正在使用 Binder 进程通信机制的进程，能够保存调用 Binder 的各个进程或线程的信息，例如线程 ID、进程 ID、Binder 状态信息等。定义 binder_proc 的具体实现代码如下所示：

```
struct binder_proc {
    //实现双向链表
    struct hlist_node proc_node;
    //线程队列、双向链表、所有的线程信息
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    //进程 ID
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;
```

```
    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;

    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo;
    //等待队列
    wait_queue_head_t wait;
    //Binder 状态
    struct binder_stats stats;
    struct list_head delivered_death;
    //最大线程
    int max_threads;
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    //默认优先级
    long default_priority;
};
```

在上述代码中，成员 proc_node 用于实现双向链表，成员 threads 用于储存所有的线程信息。

(7) binder_thread

结构体 binder_thread 用于存储每一个单独的线程的信息，表示 Binder 线程池中的一个线程。定义 binder_thread 的具体实现代码如下所示：

```
struct binder_thread {
    struct binder_proc *proc;
    struct rb_node rb_node;
    int pid;
    int looper;
    struct binder_transaction *transaction_stack;
    struct list_head todo;
    uint32_t return_error;
    uint32_t return_error2;
    wait_queue_head_t wait;
    struct binder_stats stats;
};
```

各个成员的具体说明如下所示。

- proc：表示当前线程属于哪一个 Binder 进程(binder_proc 指针)。
- rb_node：是一个红黑树节点。
- pid：表示线程的 pid。
- looper：表示线程的状态信息。
- transaction_stack：定义要接收和发送的进程和线程信息，结构体为 binder_transaction。

- todo：用于创建一个双向链表。
- return_error 和 return_error2：表示返回的错误信息代码。
- wait：是一个等待队列头结构，具体的定义代码如下所示：

```
struct binder_stats {
    int br[_IOC_NR(BR_FAILED_REPLY) + 1];
    int bc[_IOC_NR(BC_DEAD_BINDER_DONE) + 1];
    int obj_created[BINDER_STAT_COUNT];
    int obj_deleted[BINDER_STAT_COUNT];
};
```

各个成员的具体说明如下所示。

- br：用来存储 BINDER_WRITE_READ 的写操作命令协议。
- bc：用来存储 BINDER_WRITE_READ 的写操作命令协议。
- obj_created：保存 BINDER_STAT_COUNT 的对象计数，当创建一个对象时，需要同时调用该成员来增加相应的对象计数，而 obj_deleted 则正好与之相反。

looper 表示的线程状态信息在如下枚举中定义：

```
enum {
    BINDER_LOOPER_STATE_REGISTERED  = 0x01,
    BINDER_LOOPER_STATE_ENTERED     = 0x02,
    BINDER_LOOPER_STATE_EXITED      = 0x04,
    BINDER_LOOPER_STATE_INVALID     = 0x08,
    BINDER_LOOPER_STATE_WAITING     = 0x10,
    BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

上述枚举主要包括的状态信息有：注册、进入、退出、销毁、等待、需要返回。

(8)　binder_transaction

结构体 binder_transaction 的功能是中转请求和返回结果，并保存接收和要发送的进程信息。定义结构体 binder_transaction 的具体实现代码如下所示：

```
struct binder_transaction {
    int debug_id; //调试相关
    struct binder_work work;
    struct binder_thread *from;
    struct binder_transaction *from_parent;
    struct binder_proc *to_proc;
    struct binder_thread *to_thread;
    struct binder_transaction *to_parent;
    unsigned need_reply : 1;
    struct binder_buffer *buffer;
    unsigned int  code;
    unsigned int  flags;
    long priority;
    long saved_priority;
    uid_t  sender_euid;
};
```

上述成员的具体说明如下所示。

- work：是一个 binder_work。
- from 和 to_thread：都是一个 binder_thread 对象，用于表示接收和要发送的进程信息。
- from_parent 和 to_thread：接收和发送进程信息的父节点。
- to_proc：是一个 binder_proc 类型的结构体，还包括 flags、need_reply、优先级(priority)等数据。
- sender_euid：Linux 系统中的每个进程都有两个 ID，即用户 ID 和有效用户 ID，UID 一般表示进程的创建者(属于哪个用户创建)，EUID 表示进程对于文件和资源的访问权限。sender_euid 表示要发送进程对文件和资源的操作权限。

另外，在结构体 binder_transaction 中，还包含了类型类 inder_buffer 的一个 buffer，用来表示 binder 的缓冲区信息。inder_buffer 在前面已经进行了讲解。

(9) binder_write_read

结构体 binder_write_read 的功能是表示在进程之间的通信过程中传输的数据，数据包中有一个 cmd 域，用于区分不同的请求。定义结构体 binder_write_read 的实现代码如下所示：

```
struct binder_write_read {
    signed long    write_size;    /* bytes to write */
    signed long    write_consumed; /* bytes consumed by driver */
    unsigned long  write_buffer;
    signed long    read_size;     /* bytes to read */
    signed long    read_consumed;  /* bytes consumed by driver */
    unsigned long  read_buffer;
};
```

各个成员的具体说明如下所示。

- write_size 和 read_size：分别表示写入和读取的数据的大小。
- write_consumed 和 read_consumed：分别表示被消耗的写数据和读数据的大小。

当 Binder 驱动找到处理此事件的进程之后，Binder 驱动就会把需要处理的事件的任务放在读缓冲(binder_write_read)里，返回给这个服务线程，该服务线程则会执行指定命令的操作；处理请求的线程把数据交给合适的对象来执行预定操作，然后把返回结果同样用结构 binder_transaction_data 进行封装，以写命令的方式传回给 Binder 驱动，并将此数据放在一个读缓冲(binder_write_read)里，返回给正在等待结果的原进程(线程)，这样就完成了一次通信。

(10) BinderDriverCommandProtocol

结构体 binder_write_read 包含的命令在 BinderDriverCommandProtocol 中定义，具体代码如下所示：

```
enum BinderDriverCommandProtocol {
    BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
    BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
    /*
    * binder transaction data: the sent command.
    */
    BC_ACQUIRE_RESULT = _IOW('c', 2, int),
    /*
```

```
* not currently supported
* int:  0 if the last BR ATTEMPT ACQUIRE was not successful.
* Else you have acquired a primary reference on the object.
*/
BC_FREE_BUFFER = _IOW('c', 3, int),
/*
* void *: ptr to transaction data received on a read
*/
BC_INCREFS = _IOW('c', 4, int),
BC_ACQUIRE = _IOW('c', 5, int),
BC_RELEASE = _IOW('c', 6, int),
BC_DECREFS = _IOW('c', 7, int),
/*
* int: descriptor
*/
BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
/*
* void *: ptr to binder
* void *: cookie for binder
*/
BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
/*
* not currently supported
* int: priority
* int: descriptor
*/
BC_REGISTER_LOOPER = _IO('c', 11),
/*
* No parameters.
* Register a spawned looper thread with the device.
*/
BC_ENTER_LOOPER = _IO('c', 12),
BC_EXIT_LOOPER = _IO('c', 13),
/*
* No parameters.
* These two commands are sent as an application-level thread
* enters and exits the binder loop, respectively.  They are
* used so the binder can have an accurate count of the number
* of looping threads it has available.
*/
BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
/*
* void *: ptr to binder
* void *: cookie
*/
BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
/*
* void *: ptr to binder
```

```
    * void *: cookie
    */
    BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
    /*
    * void *: cookie
    */
};
```

在上述枚举命令成员中，重要的是 BC_TRANSACTION 和 BC_REPLY 命令，被作为发送操作的命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

(11) BinderDriverReturnProtocol

在枚举 BinderDriverReturnProtocol 中定义了读操作命令协议，具体实现代码如下所示：

```
enum BinderDriverReturnProtocol {
    BR_ERROR = _IOR('r', 0, int),
    /*
    * int: error code
    */
    BR_OK = _IO('r', 1),
    /* No parameters! */
    BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
    BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
    /*
    * binder_transaction_data: the received command.
    */
    BR_ACQUIRE_RESULT = _IOR('r', 4, int),
    /*
    * not currently supported
    * int: 0 if the last bcATTEMPT_ACQUIRE was not successful.
    * Else the remote object has acquired a primary reference.
    */
    BR_DEAD_REPLY = _IO('r', 5),
    /*
    * The target of the last transaction (either a bcTRANSACTION or
    * a bcATTEMPT_ACQUIRE) is no longer with us.  No parameters.
    */
    BR_TRANSACTION_COMPLETE = _IO('r', 6),
    /*
    * No parameters... always refers to the last transaction requested
    * (including replies). Note that this will be sent even for
    * asynchronous transactions.
    */
    BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
    BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
    BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
    BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
    /*
    * void *:     ptr to binder
```

```
	* void *: cookie for binder
	*/
	BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
	/*
	* not currently supported
	* int: priority
	* void *: ptr to binder
	* void *: cookie for binder
	*/
	BR_NOOP = _IO('r', 12),
	/*
	* No parameters.  Do nothing and examine the next command.  It exists
	* primarily so that we can replace it with a BR_SPAWN_LOOPER command.
	*/
	BR_SPAWN_LOOPER = _IO('r', 13),
	/*
	* No parameters.  The driver has determined that a process has no
	* threads waiting to service incoming transactions.  When a process
	* receives this command, it must spawn a new service thread and
	* register it via bcENTER_LOOPER.
	*/
	BR_FINISHED = _IO('r', 14),
	/*
	* not currently supported
	* stop threadpool thread
	*/
	BR_DEAD_BINDER = _IOR('r', 15, void *),
	/*
	* void *: cookie
	*/
	BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
	/*
	* void *: cookie
	*/
	BR_FAILED_REPLY = _IO('r', 17),
	/*
	* The the last transaction (either a bcTRANSACTION or
	* a bcATTEMPT_ACQUIRE) failed (e.g. out of memory).  No parameters.
	*/
};
```

在上述命令中，BC_TRANSACTION 和 BC_REPLY 命令被作为发送操作命令，其数据参数都是 binder_transaction_data 结构体。其中，前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

(12) binder_ptr_cookie 和 binder_transaction_data

binder_ptr_cookie 和 binder_transaction_data 是两个比较重要的结构体，其中 binder_ptr_cookie 表示一个 Binder 实体对象或 Service 组件的死亡接收通知，具体定义代码如下所示：

```
struct binder_ptr_cookie {
    void *ptr;
    void *cookie;
};
```

而 binder_transaction_data 表示在通信过程中传递的数据，具体定义代码如下所示：

```
struct binder_transaction_data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
    * identifying the target and contents of the transaction.
    */
    union {
        size_t  handle; /* target descriptor of command transaction */
        void    *ptr;   /* target descriptor of return transaction */
    } target;
    void          *cookie;      /* target object cookie */
    unsigned int    code;         /* transaction command */
    /* General information about the transaction. */
    unsigned int    flags;
    pid_t          sender_pid;
    uid_t          sender_euid;
    size_t         data_size;     /* number of bytes of data */
    size_t         offsets_size;   /* number of bytes of offsets */
    /* If this transaction is inline, the data immediately
    * follows here; otherwise, it ends with a pointer to
    * the data buffer.
    */
    union {
        struct {
            /* transaction data */
            const void    *buffer;
            /* offsets from buffer to flat_binder_object structs */
            const void    *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

(13) flat_binder_object

在 Android 系统中，将在进程之间传递的数据称为 Binder 对象，即 Binder Object。Binder 对象在对应的源码中使用结构体 flat_binder_object 来表示，具体代码如下所示：

```
struct flat_binder_object {
    /* 8 bytes for large_flat_header. */
    unsigned long       type;
    unsigned long       flags;
    /* 8 bytes of data */
    union {
        void           *binder;       /* local object */
        signed long    handle;         /* remote object */
```

```
    };
    /* extra data associated with local object */
    void                 *cookie;
};
```

各个成员的具体说明如下所示。

- type：描述了 Binder 的类型，传输的数据是一个复用数据联合体。对于 Binder 类型来说，数据是一个 Binder 本地对象。
- handle：是一个远程的 handle 句柄。假如 A 有个对象 O，对于 A 来说，O 就是一个本地的 Binder 对象；如果 B 想访问 A 的 O 对象，对于 B 来说，O 就是一个 handle。所以 handle 和 Binder 都指向 O。
- cookie：如果是本地对象，Binder 还可以带有额外的数据，这些数据将被保存到 cookie 字段中。
- flags：表示传输方式，比如同步和异步等，其值同样使用一个 enum 来表示，具体定义代码如下所示：

```
enum transaction_flags {
    TF_ONE_WAY     = 0x01, /* this is a one-way call: async, no return */
    TF_ROOT_OBJECT  = 0x04, /* contents are the component's root object */
    TF_STATUS_CODE  = 0x08, /* contents are a 32-bit status code */
    TF_ACCEPT_FDS   = 0x10, /* allow replies with file descriptors */
};
```

6.2.2　分析设备初始化

我们可以在文件 binder.c 中找到该初始化函数 binder_init，具体定义代码如下所示：

```
static int __init binder_init(void)
{
    int ret;
    binder_deferred_workqueue = create_singlethread_workqueue("binder");
    if (!binder_deferred_workqueue)
        return -ENOMEM;
    binder_debugfs_dir_entry_root = debugfs_create_dir("binder", NULL);
    if (binder_debugfs_dir_entry_root)
        binder_debugfs_dir_entry_proc =
          debugfs_create_dir("proc", binder_debugfs_dir_entry_root);
    ret = misc_register(&binder_miscdev);
    if (binder_debugfs_dir_entry_root) {
        debugfs_create_file("state",
                            S_IRUGO,
                            binder_debugfs_dir_entry_root,
                            NULL,
                            &binder_state_fops);
        debugfs_create_file("stats",
                            S_IRUGO,
                            binder_debugfs_dir_entry_root,
```

```
                            NULL,
                            &binder stats fops);
        debugfs_create_file("transactions",
                            S_IRUGO,
                            binder_debugfs_dir_entry_root,
                            NULL,
                            &binder_transactions_fops);
        debugfs_create_file("transaction_log",
                            S_IRUGO,
                            binder_debugfs_dir_entry_root,
                            &binder_transaction_log,
                            &binder_transaction_log_fops);
        debugfs_create_file("failed_transaction_log",
                            S_IRUGO,
                            binder_debugfs_dir_entry_root,
                            &binder_transaction_log_failed,
                            &binder_transaction_log_fops);
    }
    return ret;
}
```

binder_init 是 Binder 驱动的初始化函数，实现时，需要调用设备驱动接口。Android Binder 设备驱动接口函数是 device_initcall，使用 module_init 和 module_exit 是为了同时兼容支持静态编译的驱动模块(buildin)和动态编译的驱动模块(module)。Binder 使用 device_initcall 的目的就是不让 Binder 驱动支持动态编译，而且需要在内核(Kernel)做镜像。initcall 用于注册进行初始化的函数，如果的确需要将 Binder 驱动修改为动态的内核模块，可以直接将 device_initcall 修改为 module_init，并增加 module_exit 的驱动卸载接口函数。

在注册 Binder 驱动为 Misc 设备时，指定了 Binder 驱动的 miscdevice，具体实现代码如下所示：

```
static struct miscdevice binder_miscdev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "binder",
    .fops = &binder_fops
};
```

Binder 设备的主设备号为 10，此设备号是动态获得的，各个参数的具体说明如下所示。

- MISC_DYNAMIC_MINOR：.minor 被设置为此类型的动态获得设备号。
- .name：表示设备名称。
- file_operations：指定了该设备的 file_operations 结构体，定义如代码如下所示。

```
static struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
```

```
    .flush = binder_flush,
    .release = binder_release,
};
```

在 Android 系统中，任何驱动程序都具备向用户空间的程序提供操作接口的功能。这个接口是一个标准接口，Android Binder 驱动提供了操作设备文件(/dev/binder)的接口。正如 binder_fops 所描述的 file_operations 结构体一样，其中主要包括了 binder_poll、binder_ioctl、binder_mmap、binder_open、binder_flush、binder_release 等标准操作接口。

6.2.3　打开Binder设备文件

在 Android 系统的 Binder 机制中，函数 binder_open 的功能是打开 Binder 设备文件/dev/binder。在 Android 系统中，底层驱动的任何一个进程及线程都可以打开一个 Binder 设备，其打开过程的实现如代码如下所示：

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;
    binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
                 current->group_leader->pid, current->pid);
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);
    proc->tsk = current;
    INIT_LIST_HEAD(&proc->todo);
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    binder_lock(__func__);
    binder_stats_created(BINDER_STAT_PROC);
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    INIT_LIST_HEAD(&proc->delivered_death);
    filp->private_data = proc;
    binder_unlock(__func__);
    if (binder_debugfs_dir_entry_proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        proc->debugfs_entry = debugfs_create_file(strbuf, S_IRUGO,
                   binder_debugfs_dir_entry_proc, proc, &binder_proc_fops);
    }
    return 0;
}
```

函数 binder_open 的具体实现流程如下所示。

(1)　创建并分配一个 binder_proc 空间来保存 Binder 数据。

(2)　增加当前线程/进程的引用计数，给 binder_proc 的 tsk 字段赋值。

(3)　实现 binder_proc 队列的初始化，主要包括：

- 使用 INIT_LIST_HEAD 初始化链表头 todo。
- 使用 init_waitqueue_head 初始化等待队列 wait。
- 设置默认优先级(default_priority)为当前进程的 nice 值(通过 task_nice 得到当前进程的 nice 值)。

(4) 增加 BINDER_STAT_PROC 的对象计数，并通过 hlist_add_head 把创建的 binder_proc 对象添加到全局的 binder_proc 哈希表中，这样，任何一个进程就都可以访问到其他进程的 binder_proc 对象。

(5) 把当前进程(或线程)的线程组的 pid(pid 指向线程 id)赋值给 proc 的 pid 字段，同时把创建的 binder_proc 对象指针赋值给 filp 的 private_data 对象并保存起来。

(6) 在 binder proc 目录中创建只读文件/proc/binder/proc/$pid，功能是输出当前 binder proc 对象的状态。文件名以 pid 命名，但是该 pid 字段并不是当前进程/线程的 id，而是线程组的 pid，表示是线程组中第一个线程的 pid(因为我们上面是将 current->group_leader->pid 赋值给该 pid 字段的)。并且在创建该文件时也指定了操作该文件的函数接口为 binder_read_proc_proc，此函数的参数表示创建的 binder_proc 对象 proc。

再看函数 binder_release，此函数与函数 binder_open 的功能相反。当 Binder 驱动退出时，通过函数 binder_release 来释放在打开以及其他操作过程中分配的空间，并且同时清理相关的数据信息。函数 binder_release 的具体实现代码如下所示：

```
static int binder release(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc = filp->private_data;
    debugfs_remove(proc->debugfs_entry);
    binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
    return 0;
}
```

在上述代码中，首先获取使用 private_data 数据的权限，找到当前进程、线程的 pid，这样，可以得到在 open 过程中创建的以 pid 命名的用来输出当前 binder_proc 对象的状态的只读文件；然后调用函数 remove_proc_entry 实现删除操作；最后通过函数 binder_defer_work 和其参数 BINDER_DEFERRED_RELEASE 释放整个 binder_proc 对象的数据和分配的空间。

6.2.4 内存映射

在 Android 系统中，当打开 Binder 设备文件/dev/binder 后，需要调用函数 mmap 把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存了。在 Binder 设备中，对内存的映射操作是有限制的，比如 Binder 不能映射具有写权限的内存区域，最大能映射 4MB 的内存区域等。在 Android 系统中，大多数设备本身具有设备映射的设备内存，或者是在驱动初始化时由 vmalloc 或 kmalloc 等内核内存函数分配的，在 mmap 操作时分配 Binder 的设备内存。

(1) 函数 mmap 实现分配功能的流程如下所示。

① 在内核虚拟映射表上获取一个可以使用的区域。

② 分配物理页，并把物理页映射到获取的虚拟空间上。

③　每个进程/线程只能执行一次映射操作，后面的操作都会返回错误。

(2)　函数 mmap 的具体实现流程如下所示。

①　检查内存映射条件，包括映射内存大小(4MB)、flags、是否是第一次 mmap 等。

②　获得地址空间，并把此空间的地址记录在进程信息(buffer)中。

③　分配物理页面(pages)并记录下来。

④　将 buffer 插入到进程信息的 buffer 列表中。

⑤　调用函数 binder_update_page_range，将分配的物理页面和 vm 空间对应起来。

⑥　调用函数 binder_insert_free_buffer，把进程中的 buffer 插入到进程信息中。

函数 mmap 的具体实现代码如下所示：

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int ret;
    struct vm_struct *area;
    struct binder_proc *proc = filp->private_data;
    const char *failure_string;
    struct binder_buffer *buffer;

    if ((vma->vm_end - vma->vm_start) > SZ_4M)
        vma->vm_end = vma->vm_start + SZ_4M;

    binder_debug(BINDER_DEBUG_OPEN_CLOSE,
                 "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
                 proc->pid, vma->vm_start, vma->vm_end,
                 (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
                 (unsigned long)pgprot_val(vma->vm_page_prot));

    if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
        ret = -EPERM;
        failure_string = "bad vm_flags";
        goto err_bad_arg;
    }
    vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

    mutex_lock(&binder_mmap_lock);
    if (proc->buffer) {
        ret = -EBUSY;
        failure_string = "already mapped";
        goto err_already_mapped;
    }

    area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
    if (area == NULL) {
        ret = -ENOMEM;
        failure_string = "get_vm_area";
        goto err_get_vm_area_failed;
    }
```

```
    proc->buffer = area->addr;
    proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
    mutex_unlock(&binder_mmap_lock);

#ifdef CONFIG_CPU_CACHE_VIPT
    if (cache_is_vipt_aliasing()) {
        while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
            printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n",
              proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
            vma->vm_start += PAGE_SIZE;
        }
    }
#endif
    proc->pages = kzalloc(sizeof(proc->pages[0])
      * ((vma->vm_end - vma->vm_start) / PAGE_SIZE), GFP_KERNEL);
    if (proc->pages == NULL) {
        ret = -ENOMEM;
        failure_string = "alloc page array";
        goto err_alloc_pages_failed;
    }
    proc->buffer_size = vma->vm_end - vma->vm_start;

    vma->vm_ops = &binder_vm_ops;
    vma->vm_private_data = proc;

    if (binder_update_page_range(proc, 1, proc->buffer,
      proc->buffer + PAGE_SIZE, vma)) {
        ret = -ENOMEM;
        failure_string = "alloc small buf";
        goto err_alloc_small_buf_failed;
    }
    buffer = proc->buffer;
    INIT_LIST_HEAD(&proc->buffers);
    list_add(&buffer->entry, &proc->buffers);
    buffer->free = 1;
    binder_insert_free_buffer(proc, buffer);
    proc->free_async_space = proc->buffer_size / 2;
    barrier();
    proc->files = get_files_struct(proc->tsk);
    proc->vma = vma;
    proc->vma_vm_mm = vma->vm_mm;

    /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n",
            proc->pid, vma->vm_start, vma->vm_end, proc->buffer);*/
    return 0;

err_alloc_small_buf_failed:
    kfree(proc->pages);
    proc->pages = NULL;
```

```
err_alloc_pages_failed:
    mutex_lock(&binder_mmap_lock);
    vfree(proc->buffer);
    proc->buffer = NULL;
err_get_vm_area_failed:
err_already_mapped:
    mutex_unlock(&binder_mmap_lock);
err_bad_arg:
    printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n",
             proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
    return ret;
}
```

在上述代码中，参数 vm_area_struct 是一个结构体，在 mmp 的具体实现中会非常有用。为了优化查找方法，内核专门维护了 VMA 的链表和树形结构。在结构 vm_area_struct 中，很多成员函数都是用来维护这个树形结构的。VMA 的功能是管理进程地址空间中不同区域的数据结构。该函数首先对内存映射进行检查，主要包括映射内存的大小、flags 以及是否已经映射过了，并判断其映射条件是否合法；然后，通过内核函数 get_vm_area 从系统中申请可用的虚拟内存空间，在内核中申请并保留一块连续的内核虚拟内存空间区域。接着，将 binder_proc 的用户地址偏移(即用户进程的 VMA 地址与 Binder 申请的 VMA 地址的偏差)存放到 proc->user_buffer_offset 中；再接着，使用 kzalloc 函数根据请求映射的内存空间大小，分配 Binder 的核心数据结构 binder_proc 的 pages 成员，它主要用来保存指向申请的物理页的指针；最后，为 VMA 指定了 vm_operations_struct 操作，并且将 vma->vm_private_data 指向了核心数据 proc。

到目前为止，就可以真正地开始分配物理内存(page)了。物理内存的分配工作是通过函数 binder_update_page_range 实现的，该函数主要完成如下所示的工作。

- alloc_page：分配页面。
- map_vm_area：为分配的内存做映射关系。
- vm_insert_page：把分配的物理页插入到用户 VMA 区域。

函数 binder_update_page_range 的具体实现代码如下所示：

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                                    void *start, void *end,
                                    struct vm_area_struct *vma)
{
    void *page_addr;
    unsigned long user_page_addr;
    struct vm_struct tmp_area;
    struct page **page;
    struct mm_struct *mm;

    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                 "binder: %d: %s pages %p-%p\n", proc->pid,
                 allocate ? "allocate" : "free", start, end);

    if (end <= start)
        return 0;
```

```
trace binder update page range(proc, allocate, start, end);

if (vma)
    mm = NULL;
else
    mm = get_task_mm(proc->tsk);

if (mm) {
    down_write(&mm->mmap_sem);
    vma = proc->vma;
    if (vma && mm != proc->vma_vm_mm) {
        pr_err("binder: %d: vma mm and task mm mismatch\n", proc->pid);
        vma = NULL;
    }
}

if (allocate == 0)
    goto free_range;

if (vma == NULL) {
    printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
      "map pages in userspace, no vma\n", proc->pid);
    goto err_no_vma;
}

for (page_addr=start; page_addr<end; page_addr+=PAGE_SIZE) {
    int ret;
    struct page **page array ptr;
    page = &proc->pages[(page_addr-proc->buffer)/PAGE_SIZE];

    BUG_ON(*page);
    *page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO);
    if (*page == NULL) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
          "for page at %p\n", proc->pid, page_addr);
        goto err_alloc_page_failed;
    }
    tmp_area.addr = page_addr;
    tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
    page_array_ptr = page;
    ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
    if (ret) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                      "to map page at %p in kernel\n",
                      proc->pid, page_addr);
        goto err_map_kernel_failed;
    }
    user_page_addr = (uintptr_t)page_addr + proc->user_buffer_offset;
```

```
        ret = vm_insert_page(vma, user_page_addr, page[0]);
        if (ret) {
            printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                            "to map page at %lx in userspace\n",
                            proc->pid, user_page_addr);
            goto err_vm_insert_page_failed;
        }
        /* vm_insert_page does not seem to increment the refcount */
    }
    if (mm) {
        up_write(&mm->mmap_sem);
        mmput(mm);
    }
    return 0;

free_range:
    for (page_addr = end - PAGE_SIZE; page_addr >= start;

    page_addr -= PAGE_SIZE) {
        page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
        if (vma)
            zap_page_range(vma,
              (uintptr_t)page_addr + proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
        unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);

err_map_kernel_failed:
        __free_page(*page);
        *page = NULL;

err_alloc_page_failed:
        ;
    }

err_no_vma:
    if (mm) {
        up_write(&mm->mmap_sem);
        mmput(mm);
    }
    return -ENOMEM;
}
```

其中，vm_operations_struct 只包括了一个打开操作和一个关闭操作，具体的定义代码如下所示：

```
static struct vm_operations_struct binder_vm_ops = {
    .open = binder_vma_open,
    .close = binder_vma_close,
};
```

6.2.5 释放物理页面

在 Android 系统的 Binder 机制中，函数 binder_insert_free_buffer 的功能是把进程中的 buffer 插入到进程信息中。也就是说，通过此函数能够将一个空闲内核缓冲区加入到进程中的空闲内核缓冲区的红黑树中。函数 binder_insert_free_buffer 的具体实现代码如下所示：

```
static void binder_insert_free_buffer(struct binder_proc *proc,
                              struct binder_buffer *new_buffer)
{
    struct rb_node **p = &proc->free_buffers.rb_node;
    struct rb_node *parent = NULL;
    struct binder_buffer *buffer;
    size_t buffer_size;
    size_t new_buffer_size;
    BUG_ON(!new_buffer->free);
    new_buffer_size = binder_buffer_size(proc, new_buffer);
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                 "binder: %d: add free buffer, size %zd, "
                 "at %p\n", proc->pid, new_buffer_size, new_buffer);
    while (*p) {
        parent = *p;
        buffer = rb_entry(parent, struct binder_buffer, rb_node);
        BUG_ON(!buffer->free);
        buffer_size = binder_buffer_size(proc, buffer);
        if (new_buffer_size < buffer_size)
            p = &parent->rb_left;
        else
            p = &parent->rb_right;
    }
    rb_link_node(&new_buffer->rb_node, parent, p);
    rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
}
```

6.2.6 分配内核缓冲区

在 Android 系统中，Binder 在使用 buffer 的时候，一次声明一个 proc(对应一个进程)的 buffer 总大小，然后分配一页并做好映射。使用时如果发现空间不足，会接着映射并把这个 buffer 拆成两个，并把剩余的继续放到 free_buffers 里面。在 Binder 驱动程序中，函数 binder_alloc_buf 的功能是分配内核缓冲区，具体代码如下所示：

```
static struct binder_buffer* binder_alloc_buf(struct binder_proc *proc,
                                    size_t data_size,
                                    size_t offsets_size, int is_async)
{
    struct rb_node *n = proc->free_buffers.rb_node;
    struct binder_buffer *buffer;
    size_t buffer_size;
```

```
struct rb_node *best_fit = NULL;
void *has_page_addr;
void *end_page_addr;
size_t size;
if (proc->vma == NULL) {
    printk(KERN_ERR "binder: %d: binder_alloc_buf, no vma\n", proc->pid);
    return NULL;
}
size = ALIGN(data_size, sizeof(void*)) + ALIGN(offsets_size, sizeof(void*));
if (size < data_size || size < offsets_size) {
    binder_user_error("binder: %d: got transaction with invalid "
                "size %zd-%zd\n", proc->pid, data_size, offsets_size);
    return NULL;
}
if (is_async
  && proc->free_async_space < size + sizeof(struct binder_buffer)) {
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                    "binder: %d: binder_alloc_buf size %zd"
                    "failed, no async space left\n", proc->pid, size);
    return NULL;
}
while (n) {
    buffer = rb_entry(n, struct binder_buffer, rb_node);
    BUG_ON(!buffer->free);
    buffer_size = binder_buffer_size(proc, buffer);
    if (size < buffer_size) {
        best_fit = n;
        n = n->rb_left;
    } else if (size > buffer_size)
        n = n->rb_right;
    else {
        best_fit = n;
        break;
    }
}
if (best_fit == NULL) {
    printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd failed, "
                "no address space\n", proc->pid, size);
    return NULL;
}
if (n == NULL) {
    buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
    buffer_size = binder_buffer_size(proc, buffer);
}
binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
             "binder: %d: binder_alloc_buf size %zd got buff"
             "er %p size %zd\n", proc->pid, size, buffer, buffer_size);
has_page_addr =
        (void*)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK);
```

```
    if (n == NULL) {
        if (size + sizeof(struct binder_buffer) + 4 >= buffer_size)
            buffer_size = size;  /* no room for other buffers */
        else
            buffer_size = size + sizeof(struct binder_buffer);
    }
    end_page_addr = (void*)PAGE_ALIGN((uintptr_t)buffer->data + buffer_size);
    if (end_page_addr > has_page_addr)
        end_page_addr = has_page_addr;
    if (binder_update_page_range(proc, 1,
      (void*)PAGE_ALIGN((uintptr_t)buffer->data), end_page_addr, NULL))
        return NULL;
    rb_erase(best_fit, &proc->free_buffers);
    buffer->free = 0;
    binder_insert_allocated_buffer(proc, buffer);
    if (buffer_size != size) {
        struct binder_buffer *new_buffer = (void*)buffer->data + size;
        list_add(&new_buffer->entry, &buffer->entry);
        new_buffer->free = 1;
        binder_insert_free_buffer(proc, new_buffer);
    }
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                "binder: %d: binder_alloc_buf size %zd got "
                "%p\n", proc->pid, size, buffer);
    buffer->data_size = data_size;
    buffer->offsets_size = offsets_size;
    buffer->async_transaction = is_async;
    if (is_async) {
        proc->free_async_space -= size + sizeof(struct binder_buffer);
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                    "binder: %d: binder_alloc_buf size %zd "
                    "async free %zd\n", proc->pid, size,
                     proc->free_async_space);
    }
    return buffer;
}
```

再看函数 binder_insert_allocated_buffer，功能是将分配的内核缓冲区添加到目标进程的已分配物理页面的内核缓冲区红黑树中。函数 binder_insert_allocated_buffer 的具体实现代码如下所示：

```
static void binder_insert_allocated_buffer(struct binder_proc *proc,
                                          struct binder_buffer *new_buffer)
{
    struct rb_node **p = &proc->allocated_buffers.rb_node;
    struct rb_node *parent = NULL;
    struct binder_buffer *buffer;
    BUG_ON(new_buffer->free);
    while (*p) {
```

```
        parent = *p;
        buffer = rb_entry(parent, struct binder_buffer, rb_node);
        BUG_ON(buffer->free);
        if (new_buffer < buffer)
            p = &parent->rb_left;
        else if (new_buffer > buffer)
            p = &parent->rb_right;
        else
            BUG();
    }
    rb_link_node(&new_buffer->rb_node, parent, p);
    rb_insert_color(&new_buffer->rb_node, &proc->allocated_buffers);
}
```

6.2.7　释放内核缓冲区

在 Android 系统中，函数 binder_free_buf 的功能是执行释放内核缓冲区的操作，具体实现代码如下所示：

```
static void binder_free_buf(struct binder_proc *proc,
                            struct binder_buffer *buffer)
{
    size_t size, buffer_size;
    //计算要释放的内核缓冲区 buffer 的大小，保存在 buffer_size 中
    buffer_size = binder_buffer_size(proc, buffer);
    //计算数据缓冲区和偏移数组缓冲区的大小，并保存在 size 中
    size = ALIGN(buffer->data_size, sizeof(void*))
      + ALIGN(buffer->offsets_size, sizeof(void*));

    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                 "binder: %d: binder_free_buf %p size %zd buffer"
                 "_size %zd\n", proc->pid, buffer, size, buffer_size);

    BUG_ON(buffer->free);
    BUG_ON(size > buffer_size);
    BUG_ON(buffer->transaction != NULL);
    BUG_ON((void*)buffer < proc->buffer);
    BUG_ON((void*)buffer > proc->buffer + proc->buffer_size);
    //检查要释放的内核缓冲区 buffer 是否用于异步事物
    if (buffer->async_transaction) {
        proc->free_async_space += size + sizeof(struct binder_buffer);

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                     "binder: %d: binder_free_buf size %zd "
                     "async free %zd\n", proc->pid, size,
                     proc->free_async_space);
    }
    //释放内核缓冲区
```

```
    binder_update_page_range(proc, 0,
              (void*)PAGE_ALIGN((uintptr_t)buffer->data),
              (void*)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK),
              NULL);
    rb_erase(&buffer->rb_node, &proc->allocated_buffers);
    buffer->free = 1;
    if (!list_is_last(&buffer->entry, &proc->buffers)) {
        struct binder_buffer *next = list_entry(buffer->entry.next,
                                    struct binder_buffer, entry);
        if (next->free) {
            rb_erase(&next->rb_node, &proc->free_buffers);
            binder_delete_free_buffer(proc, next);
        }
    }
    if (proc->buffers.next != &buffer->entry) {
        struct binder_buffer *prev = list_entry(buffer->entry.prev,
                                    struct binder_buffer, entry);
        if (prev->free) {
            binder_delete_free_buffer(proc, buffer);
            rb_erase(&prev->rb_node, &proc->free_buffers);
            buffer = prev;
        }
    }
    binder_insert_free_buffer(proc, buffer);
}
```

再看函数 buffer_start_page 和 buffer_end_page，用于计算结构体 binder_buffer 所占用的虚拟页面的地址。具体实现代码如下所示：

```
static void* buffer_start_page(struct binder_buffer *buffer)
{
    return (void*)((uintptr_t)buffer & PAGE_MASK);
}
static void* buffer_end_page(struct binder_buffer *buffer)
{
    return (void*)(((uintptr_t)(buffer + 1) - 1) & PAGE_MASK);
}
```

再看函数 binder_delete_free_buffer，功能是删除结构体 binder_buffer，具体实现代码如下所示：

```
static void binder_delete_free_buffer(struct binder_proc *proc,
                                      struct binder_buffer *buffer)
{
    struct binder_buffer *prev, *next=NULL;
    int free_page_end = 1;
    int free_page_start = 1;

    BUG_ON(proc->buffers.next == &buffer->entry);
    prev = list_entry(buffer->entry.prev, struct binder_buffer, entry);
```

```
    BUG_ON(!prev->free);
    if (buffer_end_page(prev) == buffer_start_page(buffer)) {
        free_page_start = 0;
        if (buffer_end_page(prev) == buffer_end_page(buffer))
            free_page_end = 0;
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                        "binder: %d: merge free, buffer %p "
                        "share page with %p\n", proc->pid, buffer, prev);
    }

    if (!list_is_last(&buffer->entry, &proc->buffers)) {
        next = list_entry(buffer->entry.next, struct binder_buffer, entry);
        if (buffer_start_page(next) == buffer_end_page(buffer)) {
            free_page_end = 0;
            if (buffer_start_page(next) == buffer_start_page(buffer))
                free_page_start = 0;
            binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                        "binder: %d: merge free, buffer"
                        " %p share page with %p\n", proc->pid,
                         buffer, prev);
        }
    }
    list_del(&buffer->entry);
    if (free_page_start || free_page_end) {
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                        "binder: %d: merge free, buffer %p do "
                        "not share page%s%s with with %p or %p\n",
                         proc->pid, buffer, free_page_start ? "" : " end",
                         free_page_end ? "" : " start", prev, next);
        binder_update_page_range(proc, 0, free_page_start ?
                        buffer_start_page(buffer) : buffer_end_page(buffer),
                        (free_page_end ? buffer_end_page(buffer) :
                        buffer_start_page(buffer)) + PAGE_SIZE, NULL);
    }
}
```

6.2.8　查询内核缓冲区

在 Android 系统中，函数 binder_buffer_lookup 的功能是根据一个用户空间地址查询一个内核缓冲区，具体实现代码如下所示：

```
static struct binder_buffer* binder_buffer_lookup(struct binder_proc *proc,
                                                   void __user *user_ptr)
{
    struct rb_node *n = proc->allocated_buffers.rb_node;
    //对于已经分配的buffer空间，以内存地址为索引加入红黑树allocated_buffers中
    struct binder_buffer *buffer;
    struct binder_buffer *kern_ptr;
```

```
    kern ptr =
      user_ptr - proc->user_buffer_offset - offsetof(struct binder_buffer, data);
    /* 进程 ioctl 传下来的指针并不是 binder_buffer 的地址，而直接是 binder_buffer.data 的
       地址。user_buffer_offset 用户空间和内核空间被映射区域起始地址之间的偏移 */

    while (n) {
        buffer = rb_entry(n, struct binder_buffer, rb_node);
        BUG_ON(buffer->free);

        if (kern_ptr < buffer)
            n = n->rb_left;
        else if (kern_ptr > buffer)
            n = n->rb_right;
        else
            return buffer;
    }
    return NULL;
}
```

6.3 Binder封装库

在 Android 4.3 系统中，各个层次的源码都有与 Binder 有关的具体实现。其中主要的 Binder 库由本地原生代码实现，Java 和 C++层都定义有同样功能的供应用程序使用的 Binder 接口，它们实际上都是调用原生 Binder 库的实现。在本节的内容中，将详细讲解 Binder 封装库的基本知识。

6.3.1 Binder库的实现层次

在 Android 4.3 系统中，Binder 库的各个实现层次的具体说明如下所示。

(1) Binder 驱动部分

驱动部分位于 Binder 结构的最底层(即 Linux 内核层)，有关这部分的分析已经在本章前面做过介绍了。Binder 驱动部分用于实现 Binder 的设备驱动，主要实现如下所示的功能：

- 组织 Binder 的服务节点。
- 调用 Binder 相关的处理线程。
- 完成实际的 Binder 传输。

(2) Binder Adapter 层

Binder Adapter 层是对 Binder 驱动的封装，主要功能是操作 Binder 驱动。应用程序无须直接与 Binder 驱动程序关联，关联文件包括 IPCThreadState.cpp、ProcessState.cpp 和 Parcel.cpp 中的一些内容。

Binder 核心库是 Binder 框架的核心实现，主要包括 IBinder、Binder(服务器端)和 BpBinder(客户端)。

(3)　顶层

顶层的 Binder 框架和具体的客户端/服务端都分别有 Java 和 C++两种实现方案，主要供应用程序使用，例如摄像头和多媒体，这部分通过调用 Binder 的核心库来实现。

在文件 frameworks\native\include\binder\IInterface.h 中，分别定义了定义类 IInterface、类模板 BnInterface 和类模板 BpInterface。其中类模板 BnInterface 和 BpInterface 用于实现 Service 组件和 Client 组件，具体定义代码如下所示：

```
template<typename INTERFACE>

class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>  queryLocalInterface(const String16& _descriptor);
    virtual const String16&  getInterfaceDescriptor() const;

protected:
    virtual IBinder*  onAsBinder();
};

// ----------------------------------------------------------------------

template<typename INTERFACE>

class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder*  onAsBinder();
};
```

在使用这两个模板的时候，起到了双继承的作用。使用者定义一个接口 INTERFACE，然后使用 BnInterface 和 BpInterface 两个模板，结合自己的接口，构建自己的 BnXXX 和 BpXXX 两个类。

6.3.2　类BBinder

类模板 BnInterface 继承于类 BBinder，BBinder 是服务的载体，与 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象(BBinder 类)的调用。

从 Binder 驱动的角度看，每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类，然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。

BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。

整个 BnXXXService 的类关系如图 6-2 所示。

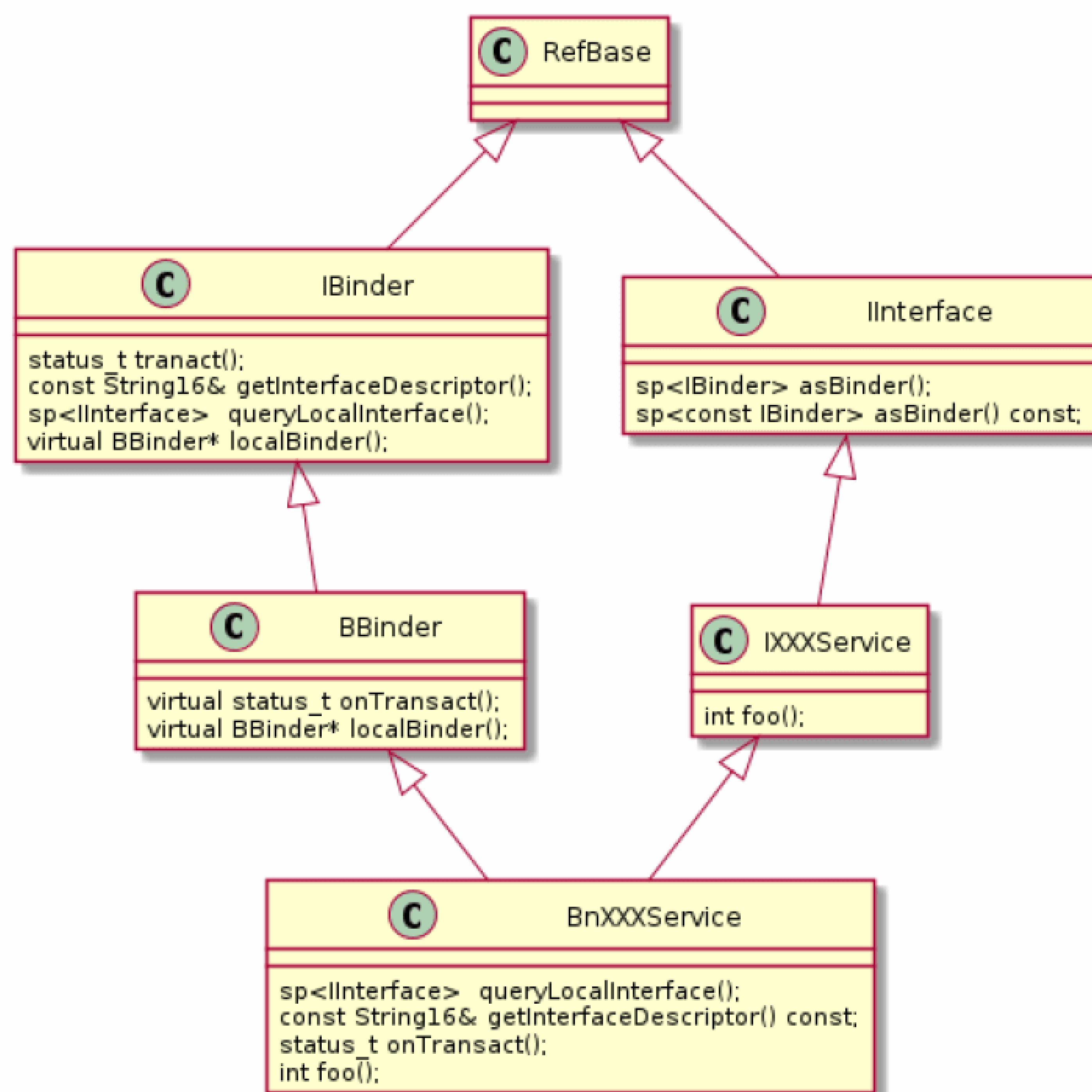


图 6-2　BnXXXService的类关系

由图 6-8 可以看出，BnXXXService 包含如下两部分。

- IXXXService：服务的主体的接口。
- BBinder：是服务的载体，与 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象(BBinder 类)的调用。

每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类。然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。

BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可以在 BnXXXService::onTransact()中完成具体的服务函数调用了。

在文件 frameworks\native\include\binder\Binder.h 中，定义类 BBinder 的代码，如下所示：

```
class BBinder : public IBinder
{
public:
    BBinder();
    virtual const String16& getInterfaceDescriptor() const;
    virtual bool       isBinderAlive() const;
    virtual status_t   pingBinder();
    virtual status_t   dump(int fd, const Vector<String16>& args);
    virtual status_t   transact(uint32_t code,
                              const Parcel &data,
                              Parcel *reply,
                              uint32_t flags=0);
```

```
    virtual status_t   linkToDeath(const sp<DeathRecipient> &recipient,
                              void *cookie=NULL,
                              uint32_t flags=0);
    virtual status_t   unlinkToDeath(const wp<DeathRecipient> &recipient,
                                  void *cookie=NULL,
                                  uint32_t flags=0,
                                  wp<DeathRecipient> *outRecipient=NULL);
    virtual void       attachObject(const void *objectID,
                                  void *object,
                                  void *cleanupCookie,
                                  object_cleanup_func func);
    virtual void*      findObject(const void *objectID) const;
    virtual void       detachObject(const void *objectID);
    virtual BBinder*   localBinder();
protected:
    virtual           ~BBinder();
    virtual status_t   onTransact(uint32_t code,
                              const Parcel &data,
                              Parcel *reply,
                              uint32_t flags=0);
private:
    BBinder(const BBinder &o);
    BBinder &operator=(const BBinder &o);
    class Extras;
    Extras *mExtras;
    void *mReserved0;
};
```

在类 BBinder 中，当一个 Binder 代理对象通过 Binder 驱动程序向一个 Binder 本地对象发出一个进程通信请求时，Binder 驱动程序会调用该 Binder 本地对象的成员函数 transact 来处理这个请求。函数 transact 在文件 frameworks\native\libs\binder\Binder.cpp 中实现，具体代码如下所示：

```
status_t BBinder::transact(
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
```

```
    }
    return err;
}
```

在上述代码中，PING_TRANSACTION 请求用来检查对象是否还存在，此处只是简单地把 pingBinder 的返回值返回给调用者，将其他的请求交给 onTransact 来处理。onTransact 是在 BBinder 中声明的一个 protected 类型的虚函数，此功能在它的子类中实现。

再看另外一个重要的成员函数 onTransact，功能是分发与业务相关的进程间通信请求。函数 onTransact 也是在文件 frameworks\native\libs\binder\Binder.cpp 中定义的，具体实现代码如下所示：

```
status_t BBinder::onTransact(
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags)
{
    switch (code) {
        case INTERFACE_TRANSACTION:
            reply->writeString16(getInterfaceDescriptor());
            return NO_ERROR;

        case DUMP_TRANSACTION: {
            int fd = data.readFileDescriptor();
            int argc = data.readInt32();
            Vector<String16> args;
            for (int i=0; i<argc && data.dataAvail()>0; i++) {
               args.add(data.readString16());
            }
            return dump(fd, args);
        }
        case SYSPROPS_TRANSACTION: {
            report_sysprop_change();
            return NO_ERROR;
        }
        default:
            return UNKNOWN_TRANSACTION;
    }
}
```

6.3.3 类BpRefBase

类模板 BpInterface 继承于类 BpRefBase，起了一个代理作用。BpRefBase 以上是业务逻辑(要实现什么功能)，BpRefBase 以下是数据传输(通过 Binder 如何将功能实现)。

在文件 frameworks\native\include\binder\Binder.h 中，定义类 BpRefBase 的代码如下所示：

```
class BpRefBase : public virtual RefBase
{
protected:
    BpRefBase(const sp<IBinder> &o);
    virtual  ~BpRefBase();
```

```
    virtual void          onFirstRef();
    virtual void          onLastStrongRef(const void *id);
    virtual bool          onIncStrongAttempted(uint32_t flags, const void *id);

    inline  IBinder*       remote()             { return mRemote; }
    inline  IBinder*       remote() const       { return mRemote; }

private:
    BpRefBase(const BpRefBase &o);
    BpRefBase&   operator=(const BpRefBase &o);

    IBinder* const         mRemote;
    RefBase::weakref_type *mRefs;
    volatile int32_t       mState;
};

}; // namespace android
```

类 BpRefBase 中的成员函数 transact 用于向运行在 Server 进程中的 Service 组件发送进程之间的通信请求，这是通过 Binder 驱动程序间接实现的。

函数 transact 的具体实现代码如下所示：

```
status_t BpBinder::transact(
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        status t status = IPCThreadState::self()->transact(
          mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD OBJECT;
}
```

6.3.4　类IPCThreadState

前面介绍的类 BBinder 和类 BpRefBase 都是通过类 IPCThreadState 和 Binder 的驱动程序交互实现的。类 IPCThreadState 在文件 frameworks\native\include\binder\IPCThreadState.h 中实现，具体实现代码如下所示：

```
class IPCThreadState
{
public:
    static  IPCThreadState*    self();
    static  IPCThreadState*    selfOrNull();  // self(), but won't instantiate

    sp<ProcessState>    process();
```

```
    status t          clearLastError();

    int              getCallingPid();
    int              getCallingUid();

    void             setStrictModePolicy(int32_t policy);
    int32_t           getStrictModePolicy() const;

    void             setLastTransactionBinderFlags(int32_t flags);
    int32_t           getLastTransactionBinderFlags() const;

    int64_t           clearCallingIdentity();
    void             restoreCallingIdentity(int64_t token);

    void             flushCommands();

    void             joinThreadPool(bool isMain = true);

    // Stop the local process.
    void             stopProcess(bool immediate = true);

    status_t          transact(int32_t handle,
                               uint32_t code, const Parcel &data,
                               Parcel *reply, uint32_t flags);

    void             incStrongHandle(int32_t handle);
    void             decStrongHandle(int32_t handle);
    void             incWeakHandle(int32 t handle);
    void             decWeakHandle(int32_t handle);
    status_t          attemptIncStrongHandle(int32_t handle);
    static  void       expungeHandle(int32_t handle, IBinder *binder);
    status_t          requestDeathNotification(int32_t handle, BpBinder *proxy);
    status_t          clearDeathNotification(int32_t handle, BpBinder *proxy);

    static  void      shutdown();

    // Call this to disable switching threads to background scheduling when
    // receiving incoming IPC calls.  This is specifically here for the
    // Android system process, since it expects to have background apps calling
    // in to it but doesn't want to acquire locks in its services while in
    // the background.
    static  void      disableBackgroundScheduling(bool disable);

private:
    IPCThreadState();
    ~IPCThreadState();

    status_t          sendReply(const Parcel &reply, uint32_t flags);
```

```
    status_t            waitForResponse(Parcel *reply,
                                        status_t *acquireResult=NULL);
    status_t            talkWithDriver(bool doReceive=true);
    status_t            writeTransactionData(int32_t cmd,
                                             uint32_t binderFlags,
                                             int32_t handle,
                                             uint32_t code,
                                             const Parcel &data,
                                             status_t *statusBuffer);
    status_t            executeCommand(int32_t command);

    void               clearCaller();

    static  void       threadDestructor(void *st);
    static  void       freeBuffer(Parcel *parcel,
                              const uint8_t *data, size_t dataSize,
                              const size_t *objects, size_t objectsSize,
                              void *cookie);

    const   sp<ProcessState>    mProcess;
    const   pid_t             mMyThreadId;
    Vector<BBinder*>     mPendingStrongDerefs;
    Vector<RefBase::weakref_type*> mPendingWeakDerefs;

    Parcel              mIn;
    Parcel              mOut;
    status_t            mLastError;
    pid_t               mCallingPid;
    uid_t               mCallingUid;
    int32_t             mStrictModePolicy;
    int32_t             mLastTransactionBinderFlags;
};

}; // namespace android
```

在类 IPCThreadState 中，成员函数用于实现数据处理。在 transact 请求中将请求的数据经过 Binder 设备发送给了 Service，Service 处理完请求后，又将结果原路返回给客户端。

函数 transact 在文件 frameworks\native\libs\binder\IPCThreadState.cpp 中定义，具体实现代码如下所示：

```
status_t IPCThreadState::transact(int32_t handle,
                              uint32_t code, const Parcel &data,
                              Parcel *reply, uint32_t flags)
{
    status_t err = data.errorCheck();
    flags |= TF_ACCEPT_FDS;
    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
```

```
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }
    return err;
}
```

6.4　初始化Java层Binder框架

虽然 Java 层 Binder 系统是 Native 层 Binder 系统的一个 Mirror，但这个 Mirror 终归还需借助 Native 层的 Binder 系统来开展工作，即它与 Native 层的 Binder 有千丝万缕的关系，故一定要在 Java 的层 Binder 正式工作前建立这种关系。

下面分析 Java 层的 Binder 框架是如何初始化的。

在 Android 系统中，函数 register_android_os_Binder 专门负责搭建 Java Binder 和 Native Binder 的交互关系。此函数在如下所示的文件中实现：

```
\frameworks\base\core\jni\android_util_Binder.cpp
```

函数 register_android_os_Binder 的具体实现代码如下所示：

```
int register_android_os_Binder(JNIEnv *env)
{
    //初始化 Java Binder 类和 Native 层的关系
    if (int_register_android_os_Binder(env) < 0)
        return -1;
    //初始化 Java BinderInternal 类和 Native 层的关系
    if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
    //初始化 Java BinderProxy 类和 Native 层的关系
    if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
    //初始化 Java Parcel 类和 Native 层的关系
    if (int_register_android_os_Parcel(env) < 0)
        return -1;
    return 0;
}
```

根据上面的代码可知，函数 register_android_os_Binder 完成了 Java 层 Binder 架构中最重要的 4 个类的初始化工作。在接下来的内容中，将详细分析 Binder 类的初始化进程。

函数 int_register_android_os_Binder 实现了 Binder 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_Binder(JNIEnv *env)
{
    jclass clazz;
    //kBinderPathName 为 Java 层中 Binder 类的全路径名，"android/os/Binder"
    clazz = env->FindClass(kBinderPathName);
    /*
    gBinderOffsets 是一个静态类对象，它专门保存 Binder 类的一些在 JNI 层中使用的信息，
    如成员函数 execTransact 的 methodID、Binder 类中成员 mObject 的 fieldID
    */
    gBinderOffsets.mClass = (jclass)env->NewGlobalRef(clazz);
```

```
    gBinderOffsets.mExecTransact =
      env->GetMethodID(clazz, "execTransact", "(IIII)Z");
    gBinderOffsets.mObject = env->GetFieldID(clazz, "mObject", "I");
    //注册 Binder 类中 native 函数的实现
    return AndroidRuntime::registerNativeMethods(
                            env, kBinderPathName,
                            gBinderMethods, NELEM(gBinderMethods));
}
```

从上面的代码可知，gBinderOffsets 对象保存了与 Binder 类相关的某些在 JNI 层中使用的信息。

下一个初始化的类是 BinderInternal，其代码在 int_register_android_os_BinderInternal 函数中。此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_BinderInternal(JNIEnv *env)
{
    jclass clazz;
    //根据 BinderInternal 的全路径名找到代表该类的 jclass 对象。全路径名为
    // "com/android/internal/os/BinderInternal"
    clazz = env->FindClass(kBinderInternalPathName);
    //gBinderInternalOffsets 也是一个静态对象，用来保存 BinderInternal 类的一些信息
    gBinderInternalOffsets.mClass = (jclass)env->NewGlobalRef(clazz);
    //获取 forceBinderGc 的 methodID
    gBinderInternalOffsets.mForceGc =
      env->GetStaticMethodID(clazz, "forceBinderGc", "()V");
    //注册 BinderInternal 类中 native 函数的实现
    return AndroidRuntime::registerNativeMethods(
                        env, kBinderInternalPathName,
                        gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

由此可见，int_register_android_os_BinderInternal 的功能与 int_register_android_os_Binder 的功能类似，主要包括以下两方面：

- 获取一些有用的 methodID 和 fieldID。这表明 JNI 层一定会向上调用 Java 层的函数。
- 注册相关类中 native 函数的实现。

函数 int_register_android_os_BinderProxy 完成了 BinderProxy 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_BinderProxy(JNIEnv *env)
{
    jclass clazz;
    clazz = env->FindClass("java/lang/ref/WeakReference");
    //gWeakReferenceOffsets 用来与 WeakReference 类打交道
    gWeakReferenceOffsets.mClass = (jclass)env->NewGlobalRef(clazz);
    //获取 WeakReference 类 get 函数的 methodID
    gWeakReferenceOffsets.mGet = env->GetMethodID(
      clazz, "get", "()Ljava/lang/Object;");
    clazz = env->FindClass("java/lang/Error");
```

```
    //gErrorOffsets 用来与 Error 类打交道
    gErrorOffsets.mClass = (jclass)env->NewGlobalRef(clazz);

    clazz = env->FindClass(kBinderProxyPathName);
    //gBinderProxyOffsets 用来与 BinderProxy 类打交道
    gBinderProxyOffsets.mClass = (jclass)env->NewGlobalRef(clazz);
    gBinderProxyOffsets.mConstructor = env->GetMethodID(clazz, "<init>", "()V");
    ... //获取 BinderProxy 的一些信息
    clazz = env->FindClass("java/lang/Class");
    //gClassOffsets 用来与 Class 类打交道
    gClassOffsets.mGetName =
      env->GetMethodID(clazz, "getName", "()Ljava/lang/String;");
    //注册 BinderProxy native 函数的实现
    return AndroidRuntime::registerNativeMethods(
      env, kBinderProxyPathName, gBinderProxyMethods,
      NELEM(gBinderProxyMethods));
}
```

根据上面的代码可知，int_register_android_os_BinderProxy 函数除了初始化 BinderProxy 类外，还获取了 WeakReference 类和 Error 类的一些信息。由此可见，BinderProxy 对象的生命周期会委托 WeakReference 来管理，故 JNI 层会获取该类 get 函数的 MethodID。

到此为止，Java Binder 几个重要成员的初始化已完成，同时在代码中定义了几个全局静态对象，分别是 gBinderOffsets、gBinderInternalOffsets 和 gBinderProxyOffsets。

第 7 章　Zygote进程、System进程和应用程序进程

在 Android 系统中，存在着 3 个十分重要的进程系统，分别是 Zygote 进程、System 进程和应用程序进程。在本章的内容中，将详细分析 Android 4.3 中这三大进程系统的基本知识，深入了解 Android 进程系统的方方面面，为读者步入本书后面知识的学习打下基础。

7.1 Zygote(孕育)进程详解

在 Android 系统中，Zygote 进程被称为“孵化”进程或“孕育”进程，功能与 Linux 系统的 fork 类似，用于“孕育”出不同的子进程。在本节的内容中，将详细讲解 Zygote 进程的基本知识。

7.1.1 Zygote基础

在 Android 系统中，如果查看进程列表的话，会发现进程 Zygote 的父进程是 init，而且它是所有应用的父进程；还有一个进程是 system_server，它的父进程是 Zygote。其实 Zygote 服务是一种 Select 服务模型，是为启动 Java 代码而生的，完成一次 androidRuntime 的打开和关闭操作。Android 系统是基于 Linux 内核的，在 Linux 系统中的所有进程都是 init 进程的子孙进程。也就是说，所有进程都是直接或者间接地由 init 进程 fork(孕育)出来的。Zygote 进程也不例外，它是在系统启动的过程中，由 init 进程创建的。Zygote 是 Android 系统的核心进程之一，被认为是 Android Framework 大家族的祖先。事实上，Zygote 正是我们平常所所说的 Java 运行环境(JVM)。从总体架构上看，Zygote 是一个简单的典型 C/S 结构。其他进程作为一个客户端向 Zygote 发出“孕育”请求，当 Zygote 接收到命令后，就“孕育”出一个 Activity 进程。具体的“孕育”过程如图 7-1 所示。

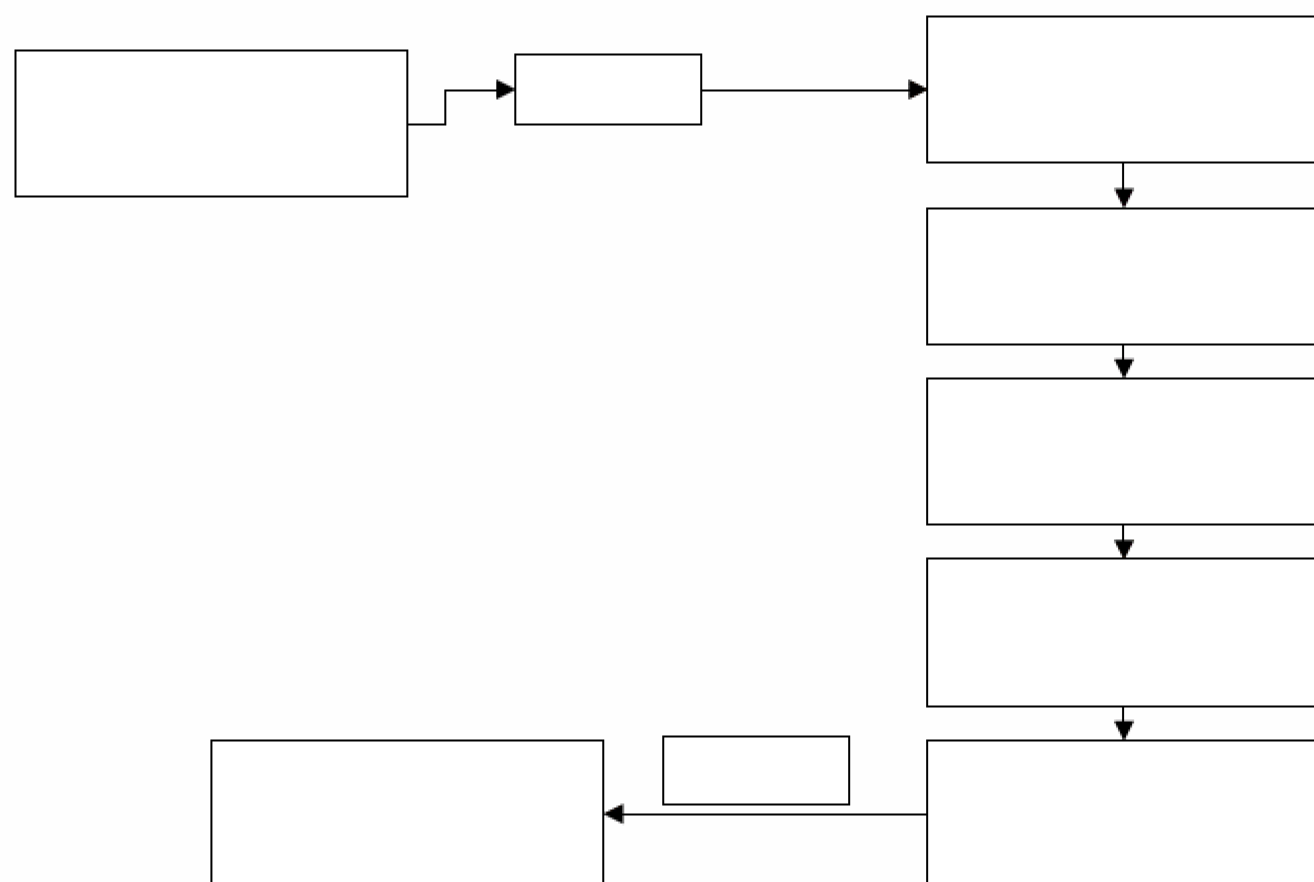


图 7-1 Zygote的“孕育”过程

在 Android 系统中，Zygote 本身是一个应用层的程序，与驱动、内核模块等没有任何关系。Zygote 系统源码的组成及其调用结构如下所示。

(1) Zygote.java：提供访问 Dalvik 的 zygote 接口，主要是包装 Linux 系统的 fork(孕育)，以建立一个新的 VM 实例进程。

(2) ZygoteConnection.java：Zygote 的套接口连接管理及其参数解析。其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote。

(3) ZygoteInit.java。Zygote 系统的 main 函数入口。

7.1.2 分析Zygote的启动过程

在 Android 4.3 源码中，在文件 system\core\rootdir\init.rc 中可以看到启动 Zygote 进程的脚本命令，具体代码如下所示：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
   socket zygote stream 666
```

通过上述代码，系统启动后会在/dev/socket 目录下看到一个名为 zygote 的文件。在上述代码中，相关关键字的具体说明如下所示。

- service：通知 init 进程创建一个名为“zygote”的进程，此 zygote 进程要执行的程序是/system/bin/app_process，后面部分需要传给 app_proces。
- socket：表示这个 zygote 进程需要一个名称为“zygote”的 socket 资源。

Zygote 最初的名字是 app_process，通过直接调用 pctrl 后把名字给改成了“zygote”。

Zygote 进程执行的程序是 system/bin/app_process，其对应的源代码在如下文件中定义：

```
frameworks/base/cmds/app_process/app_main.cpp
```

文件 app_main.cpp 的入口函数是 main， 接下来的内容中，将详细讲解启动 Zygote 进程的过程。

(1) 分析启动脚本

在文件 system\core\init\init.c 中，以服务的形式来启动 Zygote 进程。在启动初始化进程 init 时，会调用函数 service_start 来启动 Zygote。函数 service_start 的具体实现代码如下所示：

```
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid t pid;
    int needs_console;
    int n;
    char *scon = NULL;
    int rc;

        /* starting a service removes it from the disabled or reset
         * state and immediately takes it out of the restarting
         * state if it was in there
         */
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
    svc->time_started = 0;

        /* running processes require no additional work -- if
         * they're in the process of exiting, we've ensured
         * that they will immediately restart on exit, unless
         * they are ONESHOT
         */
```

```
    if (svc->flags & SVC_RUNNING) {
        return;
    }

    needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
    if (needs_console && (!have_console)) {
        ERROR("service '%s' requires console\n", svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (stat(svc->args[0], &s) != 0) {
        ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if ((!(svc->flags & SVC_ONESHOT)) && dynamic_args) {
        ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
              svc->args[0]);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (is_selinux_enabled() > 0) {
        if (svc->seclabel) {
            scon = strdup(svc->seclabel);
            if (!scon) {
                ERROR("Out of memory while starting '%s'\n", svc->name);
                return;
            }
        } else {
            char *mycon=NULL, *fcon=NULL;

            INFO("computing context for service '%s'\n", svc->args[0]);
            rc = getcon(&mycon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                return;
            }

            rc = getfilecon(svc->args[0], &fcon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                freecon(mycon);
                return;
            }

            rc = security_compute_create(
```

```
                mycon, fcon, string_to_security_class("process"), &scon);
            freecon(mycon);
            freecon(fcon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                return;
            }
        }
    }

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei=svc->envvars; ei; ei=ei->next)
            add_environment(ei->name, ei->value);

        setsockcreatecon(scon);

        for (si=svc->sockets; si; si=si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }

        freecon(scon);
        scon = NULL;
        setsockcreatecon(NULL);

        if (svc->ioprio_class != IoSchedClass_NONE) {
            if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
```

```
            ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
                getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
        }
    }

    if (needs_console) {
        setsid();
        open_console();
    } else {
        zap_stdio();
    }

#if 0
    for (n=0; svc->args[n]; n++) {
        INFO("args[%d] = '%s'\n", n, svc->args[n]);
    }
    for (n=0; ENV[n]; n++) {
        INFO("env[%d] = '%s'\n", n, ENV[n]);
    }
#endif

    setpgid(0, getpid());

  /* as requested, set our gid, supplemental gids, and uid */
    if (svc->gid) {
        if (setgid(svc->gid) != 0) {
            ERROR("setgid failed: %s\n", strerror(errno));
            _exit(127);
        }
    }
    if (svc->nr_supp_gids) {
        if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
            ERROR("setgroups failed: %s\n", strerror(errno));
            _exit(127);
        }
    }
    if (svc->uid) {
        if (setuid(svc->uid) != 0) {
            ERROR("setuid failed: %s\n", strerror(errno));
            _exit(127);
        }
    }
    if (svc->seclabel) {
        if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
            ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
            _exit(127);
        }
    }
```

```
        if (!dynamic_args) {
            if (execve(svc->args[0], (char**)svc->args, (char**)ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
            int arg_idx = svc->nargs;
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char*)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg ptrs, (char**) ENV);
        }
        _exit(127);
    }

    freecon(scon);

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

在上述代码中，每一个 Service 命令都会促使 init 进程调用 fork 函数来创建一个新的进程，在新的进程中会分析里面的 socket 选项。对于每一个 socket 选项来说，都会通过函数 create_socket 来在/dev/socket 目录下创建一个文件。由此可见，函数 service_start 起了一个解释文件 init.rc 中的 service 命令的作用。

再看函数 create_socket，功能是调用函数 socket 创建一个 Socket，使用文件描述符 fd 来描述此 Socket。函数 create_socket 的具体实现代码如下所示：

```
int create_socket(const char *name, int type, mode_t perm, uid_t uid, gid_t gid)
```

```
{
    struct sockaddr un addr;
    int fd, ret;
    char *secon;
    //调用函数 socket 创建一个 Socket，使用文件描述符 fd 来描述此 Socket
    fd = socket(PF_UNIX, type, 0);
    if (fd < 0) {
        ERROR("Failed to open socket '%s': %s\n", name, strerror(errno));
        return -1;
    }
    //为 Socket 创建一个类型为 AF_UNIX 的 Socket 地址 addr
    memset(&addr, 0 , sizeof(addr));
    addr.sun_family = AF_UNIX;
    snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s",
            name);
    ret = unlink(addr.sun_path);
    if (ret != 0 && errno != ENOENT) {
        ERROR("Failed to unlink old socket '%s': %s\n", name, strerror(errno));
        goto out_close;
    }
    secon = NULL;
    if (sehandle) {
        ret = selabel_lookup(sehandle, &secon, addr.sun_path, S_IFSOCK);
        if (ret == 0)
            setfscreatecon(secon);
    }
    ret = bind(fd, (struct sockaddr*) &addr, sizeof (addr));
    if (ret) {
        ERROR("Failed to bind socket '%s': %s\n", name, strerror(errno));
        goto out_unlink;
    }
    setfscreatecon(NULL);
    freecon(secon);
    //设置设备文件的/dev/socket/zygote 的用户 id、用户组 id 和用户权限
    chown(addr.sun_path, uid, gid);
    chmod(addr.sun_path, perm);
    INFO("Created socket '%s' with mode '%o', user '%d', group '%d'\n",
        addr.sun_path, perm, uid, gid);
    return fd;
out_unlink:
    unlink(addr.sun_path);
out_close:
    close(fd);
    return -1;
}
```

再看函数 publish_socket，具体实现代码如下所示：

```
//参数 fd 是文件描述符，指向函数 create_socket 创建的 socket
static void publish_socket(const char *name, int fd)
```

```
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];
    //将宏 ANDROID_SOCKET_ENV_PREFIX 与参数 name 描述的字符串连接在一起，
    //并保存在字符串 key 中
    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
            sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

(2) 分析入口函数

Zygote 的入口函数是 main，功能是创建 AppRuntime 变量，然后调用成员函数 start 启动进程。函数 main 是在文件 frameworks\base\cmds\app_process\app_main.cpp 中定义的，具体实现代码如下所示：

```
int main(int argc, char* const argv[])
{
#ifdef __arm__
    /*
     * b/7188322 - Temporarily revert to the compat memory layout
     * to avoid breaking third party apps.
     *
     * THIS WILL GO AWAY IN A FUTURE ANDROID RELEASE.
     *
     * http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=
commitdiff;h=7dbaa466
     * changes the kernel mapping from bottom up to top-down.
     * This breaks some programs which improperly embed
     * an out of date copy of Android's linker.
     */
    char value[PROPERTY_VALUE_MAX];
    property_get("ro.kernel.qemu", value, "");
    bool is_qemu = (strcmp(value, "1") == 0);
    if ((getenv("NO_ADDR_COMPAT_LAYOUT_FIXUP")==NULL) && !is_qemu) {
        int current = personality(0xFFFFFFFF);
        if ((current & ADDR_COMPAT_LAYOUT) == 0) {
            personality(current | ADDR_COMPAT_LAYOUT);
            setenv("NO_ADDR_COMPAT_LAYOUT_FIXUP", "1", 1);
            execv("/system/bin/app_process", argv);
            return -1;
        }
    }
    unsetenv("NO_ADDR_COMPAT_LAYOUT_FIXUP");
#endif
```

```
// These are global variables in ProcessState.cpp
mArgC = argc;
mArgV = argv;

mArgLen = 0;
for (int i=0; i<argc; i++) {
    mArgLen += strlen(argv[i]) + 1;
}
mArgLen--;

AppRuntime runtime;
const char *argv0 = argv[0];

// Process command line arguments
// ignore argv[0]
argc--;
argv++;

// Everything up to '--' or first non '-' arg goes to the vm

int i = runtime.addVmArguments(argc, argv);

// Parse runtime arguments.  Stop at first unrecognized option.
bool zygote = false;
bool startSystemServer = false;
bool application = false;
const char *parentDir = NULL;
const char *niceName = NULL;
const char *className = NULL;
while (i < argc) {
    const char *arg = argv[i++];
    if (!parentDir) {
        parentDir = arg;
    } else if (strcmp(arg, "--zygote") == 0) {
        zygote = true;
        niceName = "zygote";
    } else if (strcmp(arg, "--start-system-server") == 0) {
        startSystemServer = true;
    } else if (strcmp(arg, "--application") == 0) {
        application = true;
    } else if (strncmp(arg, "--nice-name=", 12) == 0) {
        niceName = arg + 12;
    } else {
        className = arg;
        break;
    }
}
```

```
    if (niceName && *niceName) {
        setArgv0(argv0, niceName);
        set_process_name(niceName);
    }

    runtime.mParentDir = parentDir;

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit",
                startSystemServer ? "start-system-server" : "");
    } else if (className) {
        // Remainder of args get passed to startup class main()
        runtime.mClassName = className;
        runtime.mArgC = argc - i;
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
                application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

(3) 分析启动函数

Zygote的启动函数是start，功能是调用函数startVm在Zygote中创建一个虚拟机实例。函数start是在文件frameworks\base\core\jni\AndroidRuntime.cpp中定义的，具体实现代码如下所示：

```
void AndroidRuntime::start(const char *className, const char *options)
{
    ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
            className != NULL ? className : "(unknown)");

    blockSigpipe();

    /*
     * 'startSystemServer == true' means runtime is obsolete and not run from
     * init.rc anymore, so we print out the boot start event here.
     */
    if (strcmp(options, "start-system-server") == 0) {
        /* track our progress through the boot sequence */
        const int LOG_BOOT_PROGRESS_START = 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char *rootDir = getenv("ANDROID_ROOT");
```

```
    if (rootDir == NULL) {
        rootDir = "/system";
        if (!hasDir("/system")) {
            LOG_FATAL("No root directory specified, and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //const char *kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /*创建一个虚拟机实例 */
    JNIEnv *env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * 调用函数 startReg 在虚拟机实例中注册 JNI 方法
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    /*
     * We want to call main() with a String array with arguments in it.
     * At present we have two arguments, the class name and an option string.
     * Create an array to hold them.
     */
    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
    jstring optionsStr;

    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);

    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
```

```
     * not return until the VM exits.
     */
    char *slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            //调用类 com.android.internal.os.ZygoteInit 的静态成员函数 main
            //来启动 Zygote 进程
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
    free(slashClassName);
    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

在上述代码中，通过调用类 com.android.internal.os.ZygoteInit 中的函数 main 启动了 Zygote 进程。此成员函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体的实现代码如下所示：

```
public static void main(String argv[]) {
    try {
        // Start profiling the zygote initialization.
        SamplingProfilerIntegration.start();
        //调用函数 registerZygoteSocket 创建一个 socket 接口
        registerZygoteSocket();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
            SystemClock.uptimeMillis());
        preload();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
            SystemClock.uptimeMillis());
```

```
        // Finish profiling the zygote initialization.
        SamplingProfilerIntegration.writeZygoteSnapshot();

        // Do an initial gc to clean up after startup
        gc();

        // Disable tracing so that forked processes do not inherit stale tracing
        // tags from Zygote.
        Trace.setTracingEnabled(false);

        // If requested, start system server directly from Zygote
        if (argv.length != 2) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }
        //调用函数startSystemServer启动SystemServer组件
        if (argv[1].equals("start-system-server")) {
            startSystemServer();
        } else if (!argv[1].equals("")) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }

        Log.i(TAG, "Accepting command socket connections");
        //调用函数runSelectLoop进入一个无限循环
        //在前面创建的socket接口中等待ActivityManagerService请求,
        //以创建新的应用程序进程
        runSelectLoop();

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

(4) 与Zygote进程中的Socket实现连接

在Android系统中，ActivityManagerService通过函数Process.start创建一个新的进程。函数Process.start会先通过Socket连接到Zygote进程，并由 Zygote进程实现创建新应用程序进程的功能。而类Process通过函数openZygoteSocketIfNeeded来连接到Zygote进程中的Socket。函数openZygoteSocketIfNeeded在文件frameworks\base\core\java\android\os\Process.java中定义，具体实现代码如下所示：

```
private static void openZygoteSocketIfNeeded() throws ZygoteStartFailedEx {
    int retryCount;
    if (sPreviousZygoteOpenFailed) {
        retryCount = 0;
```

```
    } else {
        retryCount = 10;
    }
    for (int retry = 0;
      (sZygoteSocket==null) && (retry<(retryCount + 1)); retry++ ) {
        if (retry > 0) {
            try {
                Log.i("Zygote", "Zygote not up yet, sleeping...");
                Thread.sleep(ZYGOTE_RETRY_MILLIS);
            } catch (InterruptedException ex) {
                // should never happen
            }
        }
        try {
            sZygoteSocket = new LocalSocket();
            sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
              LocalSocketAddress.Namespace.RESERVED));
            sZygoteInputStream =
              new DataInputStream(sZygoteSocket.getInputStream());
            sZygoteWriter = new BufferedWriter(
              new OutputStreamWriter(sZygoteSocket.getOutputStream()), 256);
            Log.i("Zygote", "Process: zygote socket opened");
            sPreviousZygoteOpenFailed = false;
            break;
        } catch (IOException ex) {
            if (sZygoteSocket != null) {
                try {
                    sZygoteSocket.close();
                } catch (IOException ex2) {
                    Log.e(LOG_TAG,"I/O exception on close after exception",
                    ex2);
                }
            }
            sZygoteSocket = null;
        }
    }
    if (sZygoteSocket == null) {
        sPreviousZygoteOpenFailed = true;
        throw new ZygoteStartFailedEx("connect failed");
    }
}
```

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 registerZygoteSocket，此函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
```

```
        try {
            String env = System.getenv(ANDROID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
        } catch (RuntimeException ex) {
            throw new RuntimeException(
                    ANDROID_SOCKET_ENV + " unset or invalid", ex);
        }

        try {
            sServerSocket = new LocalServerSocket(
                    createFileDescriptor(fileDesc));
        } catch (IOException ex) {
            throw new RuntimeException(
                    "Error binding to local socket '" + fileDesc + "'", ex);
        }
    }
}
```

在上述代码中，通过文件描述符创建了 socket 接口，此文件描述符就是本书前面介绍过的文件/dev/socket/zygote，此文件描述符通过环境变量 ANDROID_SOCKET_ENV 获得。另外，由 init 进程负责解释执行系统启动脚本文件 system\core\rootdir\init.rc，而 init 进程的源代码位于文件 system/core/init/init.c 中，由函数 service_start 负责解释文件 init.rc 中的 service 命令。在 service_ start 函数中，每一个 service 命令都会促使进程 init 调用函数 fork 创建一个新的进程，在新进程中会解析里面的 socket 选项。对于每一个 socket 选项来说，全部会通过函数 create_socket 在/dev/socket 目录下创建一个 zygote 文件，然后通过函数 publish_socket 将得到的文件描述符写入到环境变量中。函数 publish_socket 的具体实现代码如下所示：

```
static void publish_socket(const char *name, int fd)
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];

    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
            sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

在上述代码中，传进的参数 name 值为“zygote”，而 ANDROID_SOCKET_ENV_PREFIX 在文件 system\core\include\cutils\sockets.h 中的定义代码为：

```
#define ANDROID_SOCKET_ENV_PREFIX    "ANDROID_SOCKET_"
```

这样就将得到的文件描述符写入到以 ANDROID_SOCKET_zygote 为名的环境变量，这个

环境变量的值为 key 值。因为函数 ZygoteInit.registerZygoteSocket 和函数 create_socket 都是运行在同一个进程中，所以函数 ZygoteInit.registerZygoteSocket 可以直接使用文件描述符来创建一个 Java 层的 LocalServerSocket 对象。如果其他进程也需要打开/dev/socket/zygote 文件以与 Zygote 进程进行通信，则必须通过文件名作为中介来连接 LocalServerSocket。

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 startSystemServer，此函数也是在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义的，具体实现代码如下所示：

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,
             1010,1018,3001,3002,3003,3006,3007",
            "--capabilities=130104352,130104352",
            "--runtime-init",
            "--nice-name=system_server",
            "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;

    int pid;

    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

        /* Request to fork the system server process */
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
    return true;
}
```

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 startSystemServer，此函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,
             1009,1010,1018, 3001,3002,3003,3006,3007",
            "--capabilities=130104352,130104352",
            "--runtime-init",
            "--nice-name=system_server",
            "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;

    int pid;

    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
        /* Request to fork the system server process */
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
    return true;
}
```

在上述代码中，Zygote 进程通过函数 forkSystemServer“孕育”了一个新的进程来启动 SystemServer 组件，返回值 pid 为 0 的位置标示新进程的执行路径，即新建进程会执行函数 handleSystemServerProcess。

函数 handleSystemServerProcess 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void handleSystemServerProcess(
  ZygoteConnection.Arguments parsedArgs)
  throws ZygoteInit.MethodAndArgsCaller {
    closeServerSocket();
    // set umask to 0077 so new files and directories will default
    // to owner-only permissions.
    Libcore.os.umask(S_IRWXG | S_IRWXO);
    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
    }
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
          parsedArgs.niceName, parsedArgs.targetSdkVersion,
          null, parsedArgs.remainingArgs);
    } else {
        /*
        * Pass the remaining arguments to SystemServer.
        */
        RuntimeInit.zygoteInit(
          parsedArgs.targetSdkVersion, parsedArgs.remainingArgs);
    }

    /* should never reach here */
}
```

上述代码调用函数 closeServerSocket 关闭了子进程，然后调用函数 RuntimeInit.zygoteInit 进一步启动 SystemServer 组件。函数 RuntimeInit.zygoteInit 在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，具体实现代码如下所示：

```
public static final void zygoteInit(int targetSdkVersion, String[] argv)
  throws ZygoteInit.MethodAndArgsCaller {
    if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

    redirectLogStreams();
    commonInit();

    //调用函数 zygoteInitNative 来执行一个 Binder 进程间通信机制的初始化工作
    //当完成这个工作后，这个进程中的 Binder 对象就可以方便地进行进程间通信了
    nativeZygoteInit();

    applicationInit(targetSdkVersion, argv);
}
```

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 runSelectLoop，功能是在前面创建的 socket 接口中进入一个无限循环，并等待 ActivityManagerService 请求创建新的应用程序进程。

函数 runSelectLoop 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void runSelectLoop() throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);

    int loopCount = GC_LOOP_COUNT;
    while (true) {
        int index;

        /*
         * Call gc() before we block in select().
         * It's work that has to be done anyway, and it's better
         * to avoid making every child do it.  It will also
         * madvise() any free memory as a side-effect.
         *
         * Don't call it every time, because walking the entire
         * heap is a lot of overhead to free a few hundred bytes.
         */
        if (loopCount <= 0) {
            gc();
            loopCount = GC_LOOP_COUNT;
        } else {
            loopCount--;
        }

        try {
            fdArray = fds.toArray(fdArray);
            index = selectReadable(fdArray);
        } catch (IOException ex) {
            throw new RuntimeException("Error in select()", ex);
        }

        if (index < 0) {
            throw new RuntimeException("Error in select()");
        } else if (index == 0) {
            ZygoteConnection newPeer = acceptCommandPeer();
            peers.add(newPeer);
            fds.add(newPeer.getFileDesciptor());
        } else {
            boolean done;

            //将数据通过Socket接口发送出去后会执行下面的语句
            // peers.get(index)得到的是一个ZygoteConnection对象，表示一个Socket连接
            //调用ZygoteConnection.runOnce函数进一步处理
            done = peers.get(index).runOnce();
```

```
            if (done) {
                peers.remove(index);
                fds.remove(index);
            }
        }
    }
}
```

通过上述代码，可以等待 ActivityManagerService 连接这个 Socket，然后，调用函数 ZygoteConnection.runOnce 创建新的应用程序。

7.2　System进程详解

在 Android 系统中，System 进程和系统服务有着重要的关系。几乎所有的 Android 核心服务都在此进程中，如 ActivityManagerService、PowerManagerService 和 WindowManagerService 等。在本节的内容中，将详细分析 Android 4.3 中的 System 系统源码，为读者步入本书后面知识的学习打下基础。

7.2.1　启动System进程前的准备工作

在 Android 系统中，通过静态类 ZygoteInit 的成员函数 handleSystemServerProcess 来启动 System 进程。具体启动过程如图 7-2 所示。



图 7-2　启动System进程前的准备工作

(1) 首先在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中获取 Zygote 进程在启动过程中创建的 Socket。其实 System 进程不需要这个 Socket，所以会调用类 ZygoteInit 的成员函数 closeServerSocket 来关闭这个 Socket。对应的代码如下所示：

```
private static void handleSystemServerProcess(
  ZygoteConnection.Arguments parsedArgs)
  throws ZygoteInit.MethodAndArgsCaller {
    //关闭这个 Socket
    closeServerSocket();
    // set umask to 0077 so new files and directories will default to
    // owner-only permissions.
    Libcore.os.umask(S_IRWXG | S_IRWXO);
    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
    }
```

```
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
          parsedArgs.niceName, parsedArgs.targetSdkVersion,
          null, parsedArgs.remainingArgs);
    } else {
        /*
        * Pass the remaining arguments to SystemServer.
        */
        //调用类RuntimeInit的静态函数zygoteInit启动System进程
        RuntimeInit.zygoteInit(
          parsedArgs.targetSdkVersion, parsedArgs.remainingArgs);
    }
    /* should never reach here */
}
```

(2) 接下来，调用类RuntimeInit的静态函数zygoteInit启动System进程，此函数在文件frameworks\base\core\java\com\android\internal\os\ZygoteInit.java中定义，具体实现代码如下：

```
public static final void zygoteInit(int targetSdkVersion, String[] argv)
  throws ZygoteInit.MethodAndArgsCaller {
    if (DEBUG)
        Slog.d(TAG, "RuntimeInit: Starting application from zygote");

    redirectLogStreams();
    //调用函数commonInit设置System进程的时区和键盘布局等信息
    commonInit();
    //调用函数nativeZygoteInit启动一个Binder线程池
    nativeZygoteInit();

    applicationInit(targetSdkVersion, argv);
}
```

7.2.2 分析SystemServer

SystemServer是Android Java的两大支柱进程之一，另一个是专门负责孵化Java进程的Zygote。如果这两大支柱中的任何一个崩溃了，都会导致Android中Java层的崩溃。如果Java层真的崩溃了，则Linux系统中的进程init会重新启动SystemServer和Zygote，以重新建立Android的Java层。在本小节的内容中，将首先纵览和分析SystemServer的源码。

(1) 分析主函数main

SystemServer是由Zygote孵化而来的一个进程，通过ps命令，可知其进程名为system_server。在DDMS中可以看到，进程system_server的进程名为system_process。SystemServer核心逻辑的入口是函数main，此入口函数在如下所示的文件中实现：

```
\frameworks\base\services\java\com\android\server\SystemServer.java
```

文件SystemServer.java的入口函数是main，具体实现代码如下所示：

```
public static void main(String[] args) {
```

```
    if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
        //如果系统时钟早于 1970 年，则设置系统时钟从 1970 年开始
        Slog.w(TAG, "System clock is before 1970; setting to 1970.");
        SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
    }

    if (SamplingProfilerIntegration.isEnabled()) {
        SamplingProfilerIntegration.start();
        timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                //SystemServer 性能统计，每小时统计一次，统计结果输出为文件
                SamplingProfilerIntegration.writeSnapshot("system_server", null);
            } // SNAPSHOT_INTERVAL 定义为 1 小时
        }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
    }

    //与 Dalvik 虚拟机相关的设置，主要是内存使用方面的控制
    dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();

    // The system server has to run all of the time, so it needs to be
    // as efficient as possible with its memory usage.
    VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);
    //加载动态库 libandroid_servers.so
    System.loadLibrary("android_servers");
    init1(args); //调用 native 的 init1 函数
}

public static final void init2() {
    Slog.i(TAG, "Entered the Android system server!");
    Thread thr = new ServerThread();
    thr.setName("android.server.ServerThread");
    thr.start();
}
```

由此可见，函数 main 首先做一些初始化工作，然后加载动态库 libandroid_servers.so，最后调用 native 的函数 init1。该函数在 libandroid_servers.so 库中实现，在如下所示的文件中定义：

```
\frameworks\base\services\jni\com_android_server_SystemServer.cpp
```

函数 init1 的具体实现代码如下所示：

```
extern "C" int system_init();
static void android_server_SystemServer_init1(JNIEnv *env, jobject clazz)
{
    system_init();          //调用上面那个用 extern 声明的 system_init 函数
}
```

而函数 system_init 在另外一个库 libsystem_server.so 中实现，在如下所示的文件中定义：

```
\frameworks\base\cmds\system_server\library\System_init.cpp
```

函数 system_init 的具体实现代码如下所示：

```
extern "C" status_t system_init()
{
    ALOGI("Entered system_init()");

    sp<ProcessState> proc(ProcessState::self());

    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p\n", sm.get());

    sp<GrimReaper> grim = new GrimReaper();
    sm->asBinder()->linkToDeath(grim, grim.get(), 0);

    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        // Start the SurfaceFlinger
        SurfaceFlinger::instantiate();
    }

    property_get("system_init.startsensorservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        // Start the sensor service
        SensorService::instantiate();
    }

    // And now start the Android runtime.  We have to do this bit
    // of nastiness because the Android runtime initialization requires
    // some of the core system services to already be started.
    // All other servers should just start the Android runtime at
    // the beginning of their processes's main(), before calling
    // the init function.
    ALOGI("System server: starting Android runtime.\n");
    AndroidRuntime *runtime = AndroidRuntime::getRuntime();

    ALOGI("System server: starting Android services.\n");
    JNIEnv *env = runtime->getJNIEnv();
    if (env == NULL) {
        return UNKNOWN_ERROR;
    }
    jclass clazz = env->FindClass("com/android/server/SystemServer");
    if (clazz == NULL) {
        return UNKNOWN_ERROR;
    }
    jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
    if (methodId == NULL) {
        return UNKNOWN_ERROR;
```

```
    }
    env->CallStaticVoidMethod(clazz, methodId);

    ALOGI("System server: entering thread pool.\n");
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
    ALOGI("System server: exiting thread pool.\n");

    return NO_ERROR;
}
```

通过上述代码可知，SystemServer 中的函数 main 通过函数 init1，从 Java 层穿越到 Native 层，实现了一些初始化工作后，又通过 JNI 从 Native 层穿越到 Java 层去调用函数 init2。函数 init2 返回后，最终又回归到 Native 层。

(2)　分析函数 init2

在文件 SystemServer.java 中，函数 init1 较简单，其实重点内容都在函数 init2 中。函数 init2 的具体实现代码如下所示：

```
public static final void init2() {
    Thread thr = new ServerThread();
    thr.setName("android.server.ServerThread");
    thr.start(); //启动一个线程，这个线程就像英雄大会一样，聚集了各路英雄
}
```

通过上述代码，将创建一个新的线程 ServerThread，该线程的 run 函数的实现代码有 600 多行，如此之长的原因是，Android 平台中众多 Service 都汇集于此。在 Android 平台中，共有七大类 43 个 Service(包括 Watchdog)。实际上，还有一些 Service 并没有在 ServerThread 的 run 函数中露面。

这七大类服务主要包括：

- Android 的核心服务，如 ActivityManagerService、WindowManager-Service 等。
- 与通信相关的服务，如 Wifi 相关服务、Telephone 相关服务。
- 与系统功能相关的服务，如 AudioService、MountService、Usb-Service 等。
- BatteryService、VibratorService 等服务。
- EntropyService、DiskStatsService、Watchdog 等相对独立的服务。
- 蓝牙服务。
- 与 UI 紧密相关的服务，如状态栏服务、通知管理服务等。

在本章后面的内容中，将详细分析其中的第 5 类服务。该类中的 Service 之间关系简单，而且功能相对独立。

第 5 类服务包括如下所示的服务。

- EntropyService：熵服务，它与随机数的生成有关。
- ClipboardService：剪贴板服务。
- DropBoxManagerService：该服务与系统运行时日志的存储与管理有关。
- DiskStatsService 和 DeviceStorageMonitorService：这两个服务用于查看和监测系统存储空间。

- SamplingProfilerService：这个服务是从 Android 4.0 新增的，功能非常简单。
- Watchdog：即看门狗，是 Android 的“老员工”了。我们在讲解 Zygote 时曾分析过。Android 2.3 以后，其内存检测功能被去掉，所以与 Android 2.2 相比显得更简单了。

7.2.3 分析EntropyService

EntropyService 是 SystemServer 启动的第一个 Service，它以 3 个小时为单位周期性地加载和保存熵池(/dev/urandom)。但是由于/dev/urandom 本身就有的安全性要比/dev/random 相对差些，所以每隔 3 小时，Android 系统在 kernel 的熵池中增加一些附加信息，这些信息对提高随机数的质量是有些帮助的。Android 会添加如下所示的额外的信息：

```
out.println("Copyright (C) 2009 The Android Open Source Project");
out.println("All Your Randomness Are Belong To Us");
out.println(START_TIME);
out.println(START_NANOTIME);
out.println(SystemProperties.get("ro.serialno"));
out.println(SystemProperties.get("ro.bootmode"));
out.println(SystemProperties.get("ro.baseband"));
out.println(SystemProperties.get("ro.carrier"));
out.println(SystemProperties.get("ro.bootloader"));
out.println(SystemProperties.get("ro.hardware"));
out.println(SystemProperties.get("ro.revision"));
out.println(System.currentTimeMillis());
out.println(System.nanoTime());
```

根据物理学基本原理，一个系统的熵越大，该系统就越不稳定。在 Android 中，目前也只有随机数常处于这种不稳定的系统中了。在 Android 系统中，SystemServer 中添加该服务的代码如下所示：

```
ServiceManager.addService("entropy", new EntropyService());
```

上述代码非常简单，从中可直接分析 EntropyService 的构造函数，此函数在文件 EntropyService.java 中定义，具体实现代码如下所示：

```
public EntropyService() {
    //调用另外一个构造函数，getSystemDir 函数返回的是/data/system 目录
    this(getSystemDir() + "/entropy.dat", "/dev/urandom");
}
public EntropyService(String entropyFile, String randomDevice) {
    this.randomDevice = randomDevice;//urandom 是 Linux 系统中产生随机数的设备
    // /data/system/entropy.dat 文件保存了系统此前的熵信息
    this.entropyFile = entropyFile;
    //下面有 4 个关键函数
    loadInitialEntropy();
    addDeviceSpecificEntropy();
    writeEntropy();
    scheduleEntropyWriter();
}
```

从以上代码中可以看出，EntropyService 构造函数中依次调用了 4 个关键函数，这 4 个函数比较简单，具体功能如下所示。

(1)　函数 loadInitialEntropy

函数 loadInitialEntropy 的功能是，将文件 entropy.dat 中的内容写到 urandom 设备，这样可增加系统的随机性。在系统中有一个 entropy pool，在刚启动系统时，该 pool 中的内容为空，会导致早期生成的随机数变得可预测。通过将 entropy.dat 数据写到该 entropy pool(这样该 pool 中的内容就不为空)中，随机数的生成就无规律可言了。函数 loadInitialEntropy 的具体实现代码如下所示：

```
private void loadInitialEntropy() {
    try {
        RandomBlock.fromFile(entropyFile).toFile(randomDevice);
    } catch (IOException e) {
        Slog.w(TAG, "unable to load initial entropy (first boot?)", e);
    }
}
```

(2)　函数 addDeviceSpecificEntropy

函数 addDeviceSpecificEntropy 的功能是，将一些与设备相关的信息写入 urandom 设备，具体实现代码如下所示：

```
private void addDeviceSpecificEntropy() {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileOutputStream(randomDevice));
        out.println("Copyright (C) 2009 The Android Open Source Project");
        out.println("All Your Randomness Are Belong To Us");
        out.println(START_TIME);
        out.println(START_NANOTIME);
        out.println(SystemProperties.get("ro.serialno"));
        out.println(SystemProperties.get("ro.bootmode"));
        out.println(SystemProperties.get("ro.baseband"));
        out.println(SystemProperties.get("ro.carrier"));
        out.println(SystemProperties.get("ro.bootloader"));
        out.println(SystemProperties.get("ro.hardware"));
        out.println(SystemProperties.get("ro.revision"));
        out.println(System.currentTimeMillis());
        out.println(System.nanoTime());
    } catch (IOException e) {
        Slog.w(TAG, "Unable to add device specific data to the entropy pool", e);
    } finally {
        if (out != null) {
            out.close();
        }
    }
}
```

由上述代码可知，即使向 urandom 的 entropy pool 中写入了固定信息，也能增加随机数生

成的随机性。从熵的角度考虑，系统的质量越大(即 pool 中的内容越多)，该系统就越不稳定。

(3) 函数 writeEntropy

函数 writeEntropy 的功能是，读取 urandom 设备的内容到 entropy.dat 文件。具体实现代码如下所示：

```
private void writeEntropy() {
    try {
        RandomBlock.fromFile(randomDevice).toFile(entropyFile);
    } catch (IOException e) {
        Slog.w(TAG, "unable to write entropy", e);
    }
}
```

(4) 函数 scheduleEntropyWriter

函数 scheduleEntropyWriter 的功能是，向 EntropyService 内部的 Handler 发送一个 ENTROPY_WHAT 消息。该消息每 3 小时发送一次。收到该消息后，EntropyService 会再次调用 writeEntropy 函数，将 urandom 设备的内容写到 entropy.dat 中。具体实现代码如下所示：

```
private void scheduleEntropyWriter() {
    mHandler.removeMessages(ENTROPY_WHAT);
    mHandler.sendEmptyMessageDelayed(ENTROPY_WHAT, ENTROPY_WRITE_PERIOD);
}
```

通过上面的分析可知，文件 entropy.dat 保存了 urandom 设备内容的快照(每 3 小时更新一次)。当系统重新启动时，EntropyService 又利用这个文件来增加系统的熵，通过这种方式使随机数的生成更加不可预测。

7.2.4 分析DropBoxManagerService

在 Android 应用中，DropBoxManagerService(DBMS)用于生成和管理系统运行时的一些日志文件。这些日志文件大多记录的是系统或某个应用程序出错时的信息。其中向 SystemServer 添加 DBMS 的代码如下所示：

```
ServiceManager.addService(Context.DROPBOX_SERVICE,       //服务名为“dropbox”
                          new DropBoxManagerService(context,
                          new File("/data/system/dropbox")));
```

(1) 分析 DBMS 构造函数

DBMS 构造函数在如下所示的文件中实现：

```
\frameworks\base\services\java\com\android\server\DropBoxManagerService.java
```

DBMS 构造函数 DropBoxManagerService 的具体实现代码如下所示：

```
public DropBoxManagerService(final Context context, File path) {
    mDropBoxDir = path; //path 指定 dropbox 目录为/data/system/dropbox

    // Set up intent receivers
    mContext = context;
```

```
    mContentResolver = context.getContentResolver();

    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_DEVICE_STORAGE_LOW);
    filter.addAction(Intent.ACTION_BOOT_COMPLETED);
    //注册一个 Broadcast 监听对象，当系统启动完毕或者设备存储空间不足时，会收到广播
    context.registerReceiver(mReceiver, filter);
    //当 Settings 数据库相应项发生变化时候，也需要告知 DBMS 进行相应的处理
    mContentResolver.registerContentObserver(
      Settings.Global.CONTENT_URI, true,
      new ContentObserver(new Handler()) {
        @Override
        public void onChange(boolean selfChange) {
            //当 Settings 数据库发生变化时候， BroadcastReceiver 的 onReceive 函数
            //将被调用。注意第二个参数为 null
            mReceiver.onReceive(context, (Intent)null);
        }
    });

    mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            if (msg.what == MSG_SEND_BROADCAST) {
                mContext.sendBroadcastAsUser((Intent)msg.obj, UserHandle.OWNER,
                  android.Manifest.permission.READ_LOGS);
            }
        }
    };

    // The real work gets done lazily in init() -- that way service creation always
    // succeeds, and things like disk problems cause individual method failures.
}

/** Unregisters broadcast receivers and any other hooks -- for test instances */
public void stop() {
    mContext.unregisterReceiver(mReceiver);
}
```

通过上述代码可知，DBMS 注册一个 BroadcastReceiver 对象，同时会监听 Settings 数据库的变动。其核心逻辑都在此 BroadcastReceiver 的 onReceive 函数中。函数 onReceive 的主要功能是，存储空间不足时，需要删除一些旧的日志文件以节省存储空间。函数 onReceive 的具体实现代码如下所示：

```
public void onReceive(Context context, Intent intent) {
    if (intent != null
      && Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
        mBooted = true;
        return;
    }
```

```
    // Else, for ACTION DEVICE STORAGE LOW:
    mCachedQuotaUptimeMillis = 0;  // Force a re-check of quota size

    // Run the initialization in the background (not this main thread).
    // The init() and trimToFit() methods are synchronized, so they still
    // block other users -- but at least the onReceive() call can finish.
    new Thread() {
        public void run() {
            try {
                init();
                trimToFit();
            } catch (IOException e) {
                Slog.e(TAG, "Can't init", e);
            }
        }
    }.start();
}
```

函数 onReceive 会在以下 3 种情况发生时被调用：

- 当系统启动完毕时，由 BOOT_COMPLETED 广播触发。
- 当设备存储空间不足时，由 DEVICE_STORAGE_LOW 广播触发。
- 当 Settings 数据库相应项发生变化时候，该函数也会被触发。

(2) 添加 dropbox 日志文件

在 Android 4.3 系统中，要想理清一个 Service，最好从它提供的服务开始进行分析。

当某个应用程序因为发生异常而崩溃(crash)时，会调用 ActivityManagerService(AMS)的函数 handleApplicationCrash，此函数在如下所示的文件中定义：

```
\frameworks\base\services\java\com\android\server\am\ActivityManagerService.java
```

函数 handleApplicationCrash 的具体实现代码如下所示：

```
public void handleApplicationCrash(IBinder app,
  ApplicationErrorReport.CrashInfo crashInfo) {
    ProcessRecord r = findAppProcess(app, "Crash");
    final String processName = app==null?
      "system_server" : (r==null? "unknown" : r.processName);

    EventLog.writeEvent(EventLogTags.AM_CRASH, Binder.getCallingPid(),
              UserHandle.getUserId(Binder.getCallingUid()), processName,
              r == null? -1 : r.info.flags,
              crashInfo.exceptionClassName,
              crashInfo.exceptionMessage,
              crashInfo.throwFileName,
              crashInfo.throwLineNumber);
    //调用 addErrorToDropBox 函数，第一个参数是一个字符串，为“crash”
    addErrorToDropBox(
      "crash", r, processName, null, null, null, null, null, crashInfo);
```

```
    crashApplication(r, crashInfo);
}
```

下面来看函数 addErrorToDropBox，此函数也在文件 ActivityManagerService.java 中实现，具体实现代码如下所示：

```
public void addErrorToDropBox(String eventType,
  ProcessRecord process, String processName, ActivityRecord activity,
  ActivityRecord parent, String subject,
  final String report, final File logFile,
  final ApplicationErrorReport.CrashInfo crashInfo) {
    // NOTE -- this must never acquire the ActivityManagerService lock,
    // otherwise the watchdog may be prevented from resetting the system.

    final String dropboxTag = processClass(process) + "_" + eventType;
    final DropBoxManager dbox =
      (DropBoxManager)mContext.getSystemService(Context.DROPBOX SERVICE);

    // Exit early if the dropbox isn't configured to accept this report type.
    if (dbox==null || !dbox.isTagEnabled(dropboxTag)) return;

    final StringBuilder sb = new StringBuilder(1024);
    appendDropBoxProcessHeaders(process, processName, sb);
    if (activity != null) {
        sb.append("Activity: ")
          .append(activity.shortComponentName).append("\n");
    }
    if (parent!=null && parent.app!=null && parent.app.pid!=process.pid) {
        sb.append("Parent-Process: ")
          .append(parent.app.processName).append("\n");
    }
    if (parent!=null && parent!=activity) {
        sb.append("Parent-Activity: ")
          .append(parent.shortComponentName).append("\n");
    }
    if (subject != null) {
        sb.append("Subject: ").append(subject).append("\n");
    }
    sb.append("Build: ").append(Build.FINGERPRINT).append("\n");
    if (Debug.isDebuggerConnected()) {
        sb.append("Debugger: Connected\n");
    }
    sb.append("\n");

    // Do the rest in a worker thread to avoid blocking the caller on I/O
    // (After this point, we shouldn't access AMS internal data structures.)
    Thread worker = new Thread("Error dump: " + dropboxTag) {
        @Override
        public void run() {
```

```
            if (report != null) {
                sb.append(report);
            }
            if (logFile != null) {
                try {
                    sb.append(FileUtils.readTextFile(
                      logFile, 128 * 1024, "\n\n[[TRUNCATED]]"));
                } catch (IOException e) {
                    Slog.e(TAG, "Error reading " + logFile, e);
                }
            }
            if (crashInfo != null && crashInfo.stackTrace != null) {
                sb.append(crashInfo.stackTrace);
            }

            String setting = Settings.Global.ERROR_LOGCAT_PREFIX + dropboxTag;
            int lines = Settings.Global.getInt(
              mContext.getContentResolver(), setting, 0);
            if (lines > 0) {
                sb.append("\n");

                // Merge several logcat streams, and take the last N lines
                InputStreamReader input = null;
                try {
                    java.lang.Process logcat =
                      new ProcessBuilder("/system/bin/logcat",
                        "-v", "time", "-b", "events", "-b", "system", "-b",
                        "main", "-t",
                        String.valueOf(lines)).redirectErrorStream (true).start();

                    try { logcat.getOutputStream().close(); }
                    catch (IOException e) {}
                    try { logcat.getErrorStream().close(); }
                    catch (IOException e) {}
                    input = new InputStreamReader(logcat.getInputStream());

                    int num;
                    char[] buf = new char[8192];
                    while ((num = input.read(buf)) > 0) sb.append(buf, 0, num);
                } catch (IOException e) {
                    Slog.e(TAG, "Error running logcat", e);
                } finally {
                    if (input != null)
                        try { input.close(); } catch (IOException e) {}
                }
            }
            dbox.addText(dropboxTag, sb.toString());
        }
    };
```

```
    if (process == null) {
        // If process is null, we are being called from some internal code
        // and may be about to die -- run this synchronously.
        worker.run();
    } else {
        worker.start();
    }
}
```

由上述代码可知，函数 addErrorToDropBox 的核心功能是生成日志内容，并调用函数 addText 将内容传给 DBMS 的功能。函数 addText 定义在如下所示的文件中：

```
\frameworks\base\core\java\android\os\DropBoxManager.java
```

在 DropBoxManager 类中，函数 addText 的实现代码如下所示：

```
public void addText(String tag, String data) {
    try { mService.add(new Entry(tag, 0, data)); } catch (RemoteException e) {}
}
```

在上述代码中，实现了 mService 和 DBMS 的交互。DBMS 对外只提供一个 add 函数实现日志添加工作，而 DBM 提供了 3 个函数，分别是 addText、addData、addFile，以方便使用。

DBM 向 DBMS 传递的数据被封装在一个 Entry 中，DBMS 中的函数 add 在文件 DropBoxManagerService.java 中定义，具体实现代码如下所示：

```
public void add(DropBoxManager.Entry entry) {
    File temp = null;
    OutputStream output = null;
    final String tag = entry.getTag();
    try {
        int flags = entry.getFlags();
        if ((flags & DropBoxManager.IS_EMPTY) != 0)
            throw new IllegalArgumentException();

        init();
        if (!isTagEnabled(tag)) return;
        long max = trimToFit();
        long lastTrim = System.currentTimeMillis();

        byte[] buffer = new byte[mBlockSize];
        InputStream input = entry.getInputStream();

        // First, accumulate up to one block worth of data in memory before
        // deciding whether to compress the data or not.
        int read = 0;
        while (read < buffer.length) {
            int n = input.read(buffer, read, buffer.length - read);
            if (n <= 0) break;
            read += n;
```

```
        }

        // If we have at least one block, compress it -- otherwise, just write
        // the data in uncompressed form.
        temp = new File(mDropBoxDir,
          "drop" + Thread.currentThread().getId() + ".tmp");
        int bufferSize = mBlockSize;
        if (bufferSize > 4096) bufferSize = 4096;
        if (bufferSize < 512) bufferSize = 512;
        FileOutputStream foutput = new FileOutputStream(temp);
        output = new BufferedOutputStream(foutput, bufferSize);
        if (read==buffer.length && ((flags & DropBoxManager.IS_GZIPPED)==0)) {
            output = new GZIPOutputStream(output);
            flags = flags | DropBoxManager.IS_GZIPPED;
        }

        do {
            output.write(buffer, 0, read);

            long now = System.currentTimeMillis();
            if (now-lastTrim > 30*1000) {
                max = trimToFit();  // In case data dribbles in slowly
                lastTrim = now;
            }

            read = input.read(buffer);
            if (read <= 0) {
                FileUtils.sync(foutput);
                output.close();  // Get a final size measurement
                output = null;
            } else {
                output.flush();  // So the size measurement is pseudo-reasonable
            }

            long len = temp.length();
            if (len > max) {
                Slog.w(TAG, "Dropping: " + tag + " (" + temp.length()
                  + " > " + max + " bytes)");
                temp.delete();
                temp = null;  // Pass temp = null to createEntry() to leave a tombstone
                break;
            }
        } while (read > 0);

        long time = createEntry(temp, tag, flags);
        temp = null;

        final Intent dropboxIntent =
          new Intent(DropBoxManager.ACTION_DROPBOX_ENTRY_ADDED);
```

```
        dropboxIntent.putExtra(DropBoxManager.EXTRA_TAG, tag);
        dropboxIntent.putExtra(DropBoxManager.EXTRA_TIME, time);
        if (!mBooted) {
            dropboxIntent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
        }
        // Call sendBroadcast after returning from this call to avoid deadlock.
        // In particular the caller may be holding the WindowManagerService lock
        // but sendBroadcast requires a lock in ActivityManagerService.
        // ActivityManagerService has been caught holding that
        // very lock while waiting for the WindowManagerService lock.
        mHandler.sendMessage(
          mHandler.obtainMessage(MSG_SEND_BROADCAST, dropboxIntent));
    } catch (IOException e) {
        Slog.e(TAG, "Can't write: " + tag, e);
    } finally {
        try { if (output != null) output.close(); } catch (IOException e) {}
        entry.close();
        if (temp != null) temp.delete();
    }
}
```

从上述代码可知，DBMS 非常爱惜“/data”分区的空间，需要考虑每一个日志文件的压缩以节省存储空间。

(3) DBMS 和 settings 数据库

在 Android 系统中，DBMS 的运行需要依赖一些配置项。其实除了 DBMS，在 SystemServer 中还有很多服务都依赖于相关的配置项，这些配置项都是通过 SettingsProvider 操作 Settings 数据库来设置和查询的。SettingsProvider 是系统中很重要的一个 APK，如果将其删除后，系统就不能正常启动了。

与系统相关的配置项都在 Settings 数据库的 Secure 表内，具体说明如下所示：

```
//用来判断是否允许记录该 tag 类型的日志文件。默认是允许生成任何 tag 类型的文件
Secure.DROPBOX_TAG_PREFIX+tag: "dropbox:"+tag
//用于控制每个日志文件的存活时间，默认是 3 天。大于 3 天的日志文件就会被删除以节省空间
Secure.DROPBOX_AGE_SECONDS: "dropbox_age_seconds"
//用于控制系统保存的日志文件个数，默认是 1000 个文件
Secure.DROPBOX_MAX_FILES: "dropbox_max_files"
//用于控制 dropbox 目录最多占存储空间容量的比例，默认是 10%
Secure.DROPBOX_QUOTA_PERCENT: "dropbox_quota_percent"
//不允许 dropbox 使用的存储空间的比例，默认是 10%，也就是 dropbox 最多只能使用 90%的空间
Secure.DROPBOX_RESERVE_PERCENT: "dropbox_reserve_percent"
//dropbox 最大能使用的空间大小，默认是 5MB
Secure.DROPBOX_QUOTA_KB: "dropbox_quota_kb"
```

读者可以利用 adb shell 进入/data/data/com.android.providers.settings/databases/目录，然后利用 sqlite3 命令操作 settings.db，通过里面的表 Secure 可以了解相关内容。不过系统中的很多选项在该表中都没有相关设置，因此实际运行时都会使用代码中设置的默认值。

7.2.5 分析DiskStatsService

在 Android 4.3 中，在如下所示的文件中实现 DiskStatsService：

```
\frameworks\base\services\java\com\android\server\DiskStatsService.java
```

文件 DiskStatsService.java 的具体实现代码如下所示：

```
import java.io.File;
import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * This service exists only as a "dumpsys" target which reports
 * statistics about the status of the disk.
 */
public class DiskStatsService extends Binder {
    private static final String TAG = "DiskStatsService";

    private final Context mContext;

    public DiskStatsService(Context context) {
        mContext = context;
    }

    @Override
    protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
        mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DUMP, TAG);

        // Run a quick-and-dirty performance test: write 512 bytes
        byte[] junk = new byte[512];
        for (int i=0; i<junk.length; i++) junk[i] = (byte) i;  // Write nonzero bytes

        File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
        FileOutputStream fos = null;
        IOException error = null;

        long before = SystemClock.uptimeMillis();
        try {
            fos = new FileOutputStream(tmp);
            fos.write(junk);
        } catch (IOException e) {
            error = e;
        } finally {
            try { if (fos != null) fos.close(); } catch (IOException e) {}
        }
```

```
        long after = SystemClock.uptimeMillis();
        if (tmp.exists()) tmp.delete();

        if (error != null) {
            pw.print("Test-Error: ");
            pw.println(error.toString());
        } else {
            pw.print("Latency: ");
            pw.print(after - before);
            pw.println("ms [512B Data Write]");
        }

        reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
        reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
        reportFreeSpace(new File("/system"), "System", pw);

        // TODO: Read /proc/yaffs and report interesting values;
        // add configurable (through args) performance test parameters.
    }

    private void reportFreeSpace(File path, String name, PrintWriter pw) {
        try {
            StatFs statfs = new StatFs(path.getPath());
            long bsize = statfs.getBlockSize();
            long avail = statfs.getAvailableBlocks();
            long total = statfs.getBlockCount();
            if (bsize <= 0 || total <= 0) {
                throw new IllegalArgumentException(
                    "Invalid stat: bsize=" + bsize + " avail=" + avail + " total=" + total);
            }

            pw.print(name);
            pw.print("-Free: ");
            pw.print(avail * bsize / 1024);
            pw.print("K / ");
            pw.print(total * bsize / 1024);
            pw.print("K total = ");
            pw.print(avail * 100 / total);
            pw.println("% free");
        } catch (IllegalArgumentException e) {
            pw.print(name);
            pw.print("-Error: ");
            pw.println(e.toString());
            return;
        }
    }
}
```

从上述代码可以看出：虽然 DiskStatsService 从 Binder 中派生，但是并没有实现任何接口，

也就是说，DiskStatsService 没有任何可调用的业务函数。但是在系统中为什么会存在这样的服务呢？要想解决这个问题，需要先了解系统中的 dumpsys 命令，此命令用于打印系统中指定服务的信息，在如下文件中定义：

```
\frameworks\native\cmds\dumpsys\dumpsys.cpp
```

文件 dumpsys.cpp 的具体实现代码如下所示：

```
#define LOG_TAG "dumpsys"

#include <utils/Log.h>
#include <binder/Parcel.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include <utils/TextOutput.h>
#include <utils/Vector.h>

#include <getopt.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

using namespace android;

static int sort_func(const String16 *lhs, const String16 *rhs)
{
    return lhs->compare(*rhs);
}

int main(int argc, char* const argv[])
{
    signal(SIGPIPE, SIG_IGN);
    sp<IServiceManager> sm = defaultServiceManager();
    fflush(stdout);
    if (sm == NULL) {
        ALOGE("Unable to get default service manager!");
        aerr << "dumpsys: Unable to get default service manager!" << endl;
        return 20;
    }

    Vector<String16> services;
    Vector<String16> args;
    if (argc == 1) {
        services = sm->listServices();
        services.sort(sort_func);
        args.add(String16("-a"));
    } else {
```

```
        services.add(String16(argv[1]));
        for (int i=2; i<argc; i++) {
            args.add(String16(argv[i]));
        }
    }

    const size_t N = services.size();

    if (N > 1) {
        // first print a list of the current services
        aout << "Currently running services:" << endl;

        for (size_t i=0; i<N; i++) {
            sp<IBinder> service = sm->checkService(services[i]);
            if (service != NULL) {
                aout << "  " << services[i] << endl;
            }
        }
    }

    for (size_t i=0; i<N; i++) {
        sp<IBinder> service = sm->checkService(services[i]);
        if (service != NULL) {
            if (N > 1) {
                aout << "------------------------------------------" << endl;
                aout << "DUMP OF SERVICE " << services[i] << ":" << endl;
            }
            int err = service->dump(STDOUT_FILENO, args);
            if (err != 0) {
                aerr << "Error dumping service info: (" << strerror(err)
                        << ") " << services[i] << endl;
            }
        } else {
            aerr << "Can't find service: " << services[i] << endl;
        }
    }

    return 0;
}
```

通过上述代码可知，dumpsys 通过 Binder 调用某个 Service 的 dump 函数。上述代码的具体实现流程如下。

(1) 先获取与 ServiceManager 进程通信的 BpServiceManager 对象。

(2) 如果输入参数的个数为 1，则先查询在 SM 中注册的所有 Service。

(3) 将 Service 排序。

(4) 指定查询某个 Service。

(5) 保存剩余参数，以后可以传给 Service 的 dump 函数。

(6) 通过 Binder 调用该 Service 的 dump 函数，将 args 也传给 dump 函数。

接下来，看文件 DiskStatsService.java 中的函数 dump，具体实现代码如下所示：

```
protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {

    mContext.enforceCallingOrSelfPermission(
      android.Manifest.permission.DUMP, TAG);

    // Run a quick-and-dirty performance test: write 512 bytes
    byte[] junk = new byte[512];
    for (int i=0; i<junk.length; i++) junk[i] = (byte)i;  // Write nonzero bytes

    File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
    FileOutputStream fos = null;
    IOException error = null;

    long before = SystemClock.uptimeMillis();
    try {
        fos = new FileOutputStream(tmp);
        fos.write(junk);
    } catch (IOException e) {
        error = e;
    } finally {
        try { if (fos != null) fos.close(); } catch (IOException e) {}
    }

    long after = SystemClock.uptimeMillis();
    if (tmp.exists()) tmp.delete();

    if (error != null) {
        pw.print("Test-Error: ");
        pw.println(error.toString());
    } else {
        pw.print("Latency: ");
        pw.print(after - before);
        pw.println("ms [512B Data Write]");
    }

    reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
    reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
    reportFreeSpace(new File("/system"), "System", pw);

    // TODO: Read /proc/yaffs and report interesting values;
    // add configurable (through args) performance test parameters.
}
```

从上述代码可知，DiskStatsService 没有实现任何业务接口，只是为了调试而存在。

7.2.6　分析DeviceStorageManagerService

在 Android 4.3 中，DeviceStorageManagerService(DSMS)用于监测系统内部存储空间的状态，添加该服务的代码如下所示：

```
//DSMS 的服务名为 devicestoragemonitor
ServiceManager.addService(DeviceStorageMonitorService.SERVICE,
  new DeviceStorageMonitorService(context));
```

在如下文件中实现 DSMS 的构造函数：

```
\frameworks\base\services\java\com\android\server\DeviceStorageMonitorService.java
```

函数 DeviceStorageMonitorService 的具体实现代码如下所示：

```
public DeviceStorageMonitorService(Context context) {
    mLastReportedFreeMemTime = 0;
    mContext = context;
    mContentResolver = mContext.getContentResolver();
    mDataFileStats = new StatFs(DATA_PATH); // 获取 data 分区的信息
    mSystemFileStats = new StatFs(SYSTEM_PATH); // 获取 system 分区的信息
    mCacheFileStats = new StatFs(CACHE_PATH); // 获取 cache 分区的信息
    //获得 data 分区的总大小
    mTotalMemory = ((long)mDataFileStats.getBlockCount()
                     * mDataFileStats.getBlockSize())/100L;
    /*
    创建 3 个 Intent，分别用于通知存储空间不足、存储空间恢复正常和存储空间满。
    由于设置了 REGISTERED_ONLY_BEFORE_BOOT 标志，这 3 个 Intent 广播只能由
    系统服务接收
    */
    mStorageLowIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_LOW);
    mStorageLowIntent.addFlags(
      Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageOkIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_OK);
    mStorageOkIntent.addFlags(
      Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageFullIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_FULL);
    mStorageFullIntent.addFlags(
      Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
    mStorageNotFullIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_NOT_FULL);
    mStorageNotFullIntent.addFlags(
      Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);

    //查询 Settings 数据库中 sys_storage_threshold_percentage 的值，默认是 10，
    //即当/data 空间只剩下 10%的时候，认为空间不足
    mMemLowThreshold = getMemThreshold();
    //查询 Settings 数据库中的 sys_storage_full_threshold_bytes 的值，默认是 1MB，
    //即当 data 分区只剩 1MB 时，就认为空间已满，剩下的这 1MB 空间保留给系统自用
    mMemFullThreshold = getMemFullThreshold();
```

```
    //检查内存
    checkMemory(true);
}
```

再来看函数 checkMemory，此函数也是在文件 DeviceStorageMonitorService.java 中定义的，具体实现代码如下所示：

```
private final void checkMemory(boolean checkCache) {
    //if the thread that was started to clear cache is still running
    //do nothing till its finished clearing cache.
    //Ideally this flag could be modified by clearCache
    //and should be accessed via a lock
    //but even if it does this test will fail now and
    //hopefully the next time this flag will be set to the correct value.
    if(mClearingCache) {
        if(localLOGV) Slog.i(TAG, "Thread already running just skip");
        //make sure the thread is not hung for too long
        long diffTime = System.currentTimeMillis() - mThreadStartTime;
        if(diffTime > (10*60*1000)) {
            Slog.w(TAG, "Thread that clears cache file seems to run for ever");
        }
    } else {
        restatDataDir();
        if (localLOGV)  Slog.v(TAG, "freeMemory=" + mFreeMem);

        //post intent to NotificationManager to display icon if necessary
        if (mFreeMem < mMemLowThreshold) {
            if (checkCache) {
                // We are allowed to clear cache files at this point to
                // try to get down below the limit, because this is not
                // the initial call after a cache clear has been attempted.
                // In this case we will try a cache clear if our free
                // space has gone below the cache clear limit.
                if (mFreeMem < mMemCacheStartTrimThreshold) {
                    // We only clear the cache if the free storage has changed
                    // a significant amount since the last time.
                    if ((mFreeMemAfterLastCacheClear-mFreeMem)
                      >= ((mMemLowThreshold-mMemCacheStartTrimThreshold)/4)) {
                        // See if clearing cache helps
                        // Note that clearing cache is asynchronous and so we do a
                        // memory check again once the cache has been cleared.
                        mThreadStartTime = System.currentTimeMillis();
                        mClearSucceeded = false;
                        clearCache();
                    }
                }
            } else {
                // This is a call from after clearing the cache.  Note
                // the amount of free storage at this point.
```

```
                mFreeMemAfterLastCacheClear = mFreeMem;
                if (!mLowMemFlag) {
                    // We tried to clear the cache, but that didn't get us
                    // below the low storage limit.  Tell the user.
                    Slog.i(TAG, "Running low on memory. Sending notification");
                    sendNotification();
                    mLowMemFlag = true;
                } else {
                    if (localLOGV) Slog.v(TAG, "Running low on memory "
                      + "notification already sent. do nothing");
                }
            }
        } else {
            mFreeMemAfterLastCacheClear = mFreeMem;
            if (mLowMemFlag) {
                Slog.i(TAG, "Memory available. Cancelling notification");
                cancelNotification();
                mLowMemFlag = false;
            }
        }
        if (mFreeMem < mMemFullThreshold) {
            if (!mMemFullFlag) {
                sendFullNotification();
                mMemFullFlag = true;
            }
        } else {
            if (mMemFullFlag) {
                cancelFullNotification();
                mMemFullFlag = false;
            }
        }
    }
    if(localLOGV) Slog.i(TAG, "Posting Message again");
    //keep posting messages to itself periodically
    postCheckMemoryMsg(true, DEFAULT_CHECK_INTERVAL);
}
```

空间不足时，DSMS 会先使用函数 clearCache 进行处理，在此函数内部会与 Package Manager-Service(简称 PKMS)进行交互。

函数 clearCache 在文件 DeviceStorageManagerService.java 中定义，具体实现代码如下：

```
private final void clearCache() {
    if (mClearCacheObserver == null) {
        // Lazy instantiation
        mClearCacheObserver = new CachePackageDataObserver();
    }
    mClearingCache = true;
    try {
        if (localLOGV) Slog.i(TAG, "Clearing cache");
```

```
        IPackageManager.Stub.asInterface(ServiceManager.getService("package"))
          .freeStorageAndNotify(mMemCacheTrimToThreshold, mClearCacheObserver);
    } catch (RemoteException e) {
        Slog.w(TAG, "Failed to get handle for PackageManger Exception: " + e);
        mClearingCache = false;
        mClearSucceeded = false;
    }
}
```

CachePackageDataObserver 是 DSMS 定义的内部类，其中的函数 onRemoveCompleted 用于重新发送消息，让 DSMS 再检测一次存储空间。函数 DeviceStorageManagerService 并没有重载 dump 函数。

7.2.7 分析SamplingProfilerService

在 Android 4.3 的源码中，添加 SamplingProfilerService 服务的实现代码如下所示：

```
ServiceManager.addService("samplingprofiler", //服务名
                        new SamplingProfilerService(context));
```

在本节的内容中，将详细分析 Android 4.3 中 SamplingProfilerService 的源码。

(1) 分析 SamplingProfilerService 构造函数

SamplingProfilerService 的构造函数在如下文件中实现：

```
\frameworks\base\services\java\com\android\server\SamplingProfilerService.java
```

在文件 SamplingProfilerService.java 中，函数 SamplingProfilerService 的具体实现代码如下所示：

```
public SamplingProfilerService(Context context) {
    //注册一个 CotentObserver，用于监测 Settings 数据库的变化
    registerSettingObserver(context);
    startWorking(context); //startWorking 函数
}
```

上述代码的核心是函数 startWorking，此函数在文件 SamplingProfilerService.java 中定义，具体实现代码如下所示：

```
private void startWorking(Context context) {
    if (LOCAL_LOGV) Slog.v(TAG, "starting SamplingProfilerService!");

    final DropBoxManager dropbox =
      (DropBoxManager)context.getSystemService(Context.DROPBOX_SERVICE);

    // before FileObserver is ready, there could have already been some snapshots
    // in the directory, we don't want to miss them
    File[] snapshotFiles = new File(SNAPSHOT_DIR).listFiles();
    for (int i=0; snapshotFiles!=null && i<snapshotFiles.length; i++) {
        handleSnapshotFile(snapshotFiles[i], dropbox);
    }
```

```
    // detect new snapshot and put it in dropbox
    // delete it afterwards no matter what happened before
    // Note: needs listening at event ATTRIB rather than CLOSE_WRITE, because we
    // set the readability of snapshot files after writing them!
    snapshotObserver = new FileObserver(SNAPSHOT_DIR, FileObserver.ATTRIB) {
        @Override
        public void onEvent(int event, String path) {
            handleSnapshotFile(new File(SNAPSHOT_DIR, path), dropbox);
        }
    };
    snapshotObserver.startWatching();

    if (LOCAL_LOGV) Slog.v(TAG, "SamplingProfilerService activated");
}
```

通过上述代码可知，SamplingProfilerService 本身并不提供性能统计的功能。统计功能是通过类 SamplingProfilerIntegration 实现的，这个类封装了一个 SamplingProfiler(由 Dalvik 虚拟机提供)对象，并提供了方便利用的函数进行性能统计。

(2)　分析 SamplingProfilerIntegration

通过使用 SamplingProfilerIntegration 可以进行性能统计。在 Android 系统中有很多重要进程都需要对性能进行分析，比如 Zygote，其相关代码在如下文件中实现：

```
\frameworks\base\core\java\com\android\internal\os\ZygoteInit.java
```

在文件 ZygoteInit.java 中，与性能分析相关的代码如下所示：

```
public static void main(String argv[]) {
    try {
        // Start profiling the zygote initialization.
        SamplingProfilerIntegration.start();

        registerZygoteSocket();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
          SystemClock.uptimeMillis());
        preload();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
          SystemClock.uptimeMillis());

        // Finish profiling the zygote initialization.
        SamplingProfilerIntegration.writeZygoteSnapshot();

        // Do an initial gc to clean up after startup
        gc();

        // If requested, start system server directly from Zygote
        if (argv.length != 2) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }
```

```
        if (argv[1].equals("start-system-server")) {
            startSystemServer();
        } else if (!argv[1].equals("")) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }

        Log.i(TAG, "Accepting command socket connections");

        if (ZYGOTE_FORK_MODE) {
            runForkMode();
        } else {
            runSelectLoopMode();
        }

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

在上述代码中，函数 start 在如下文件中实现：

```
\frameworks\base\core\java\com\android\internal\os\SamplingProfilerIntegration.java
```

函数 start 的具体实现代码如下所示：

```
public static void start() {
    if (!enabled) {      //判断是否开启性能统计
        return;
    }
    if (samplingProfiler != null) {
        Log.e(TAG, "SamplingProfilerIntegration already started at "
          + new Date(startMillis));
        return;
    }

    ThreadGroup group = Thread.currentThread().getThreadGroup();
    //创建一个 Dalvik 的 SamplingProfiler
    SamplingProfiler.ThreadSet threadSet =
      SamplingProfiler.newThreadGroupTheadSet(group);
    samplingProfiler = new SamplingProfiler(samplingProfilerDepth, threadSet);
    //启动统计
    samplingProfiler.start(samplingProfilerMilliseconds);
    startMillis = System.currentTimeMillis();
}
```

在上述代码中，使用该类的 static 语句来判断是否启动性能统计的 enable 变量由谁控制。在文件 SamplingProfilerIntegration.java 中，static 语句的实现代码如下所示：

```
static {
    samplingProfilerMilliseconds =
      SystemProperties.getInt("persist.sys.profiler_ms", 0);
    samplingProfilerDepth =
      SystemProperties.getInt ("persist.sys.profiler_depth", 4);
    if (samplingProfilerMilliseconds > 0) {
        File dir = new File(SNAPSHOT_DIR);
        dir.mkdirs();
        // the directory needs to be writable to anybody to allow file writing
        dir.setWritable(true, false);
        // the directory needs to be executable to anybody to allow file creation
        dir.setExecutable(true, false);
        if (dir.isDirectory()) {
            snapshotWriter =
              Executors.newSingleThreadExecutor(new ThreadFactory() {
                public Thread newThread(Runnable r) {
                    return new Thread(r, TAG);
                }
            });
            enabled = true;
            Log.i(TAG, "Profiling enabled. Sampling interval ms: "
              + samplingProfilerMilliseconds);
        } else {
            snapshotWriter = null;
            enabled = true;
            Log.w(TAG, "Profiling setup failed. Could not create " + SNAPSHOT_DIR);
        }
    } else {
        snapshotWriter = null;
        enabled = false;
        Log.i(TAG, "Profiling disabled.");
    }
}
```

由上述代码可知，enable 的控制在 static 语句中实现，这表明要使用性能统计，就必须重新启动要统计的进程。

当启动性能统计后，需要输出统计文件，这些功能由函数 writeZygoteSnapshot 实现。在文件 SamplingProfilerIntegration.java 中，函数 writeZygoteSnapshot 的具体实现代码如下所示：

```
public static void writeZygoteSnapshot() {
    if (!enabled) {
        return;
    }
    writeSnapshotFile("zygote", null);
    samplingProfiler.shutdown();
    samplingProfiler = null;
```

```
    startMillis = 0;
}
```

在上述代码中，调用了 writeSnapshotFile 函数，其第一个参数为 zygote，用于表示进程名。writeSnapshotFile 函数比较简单，功能就是在 shots 目录下生成一个统计文件，统计文件的名称由两部分组成，合起来就是“进程名_开始性能统计的时刻.snapshot”。另外，writeSnapshotFile 函数内部会调用 generateSnapshotHeader 函数在该统计文件的文件头部写一些特定的信息，例如版本号、编译信息等。在文件 SamplingProfilerIntegration.java 中，函数 writeSnapshotFile 的具体实现代码如下所示：

```
private static void writeSnapshotFile(String processName, PackageInfo packageInfo)
{
    if (!enabled) {
        return;
    }
    samplingProfiler.stop();
    String name = processName.replaceAll(":", ".");
    String path = SNAPSHOT_DIR + "/" + name + "-" + startMillis + ".snapshot";
    long start = System.currentTimeMillis();
    OutputStream outputStream = null;
    try {
        outputStream = new BufferedOutputStream(new FileOutputStream(path));
        PrintStream out = new PrintStream(outputStream);
        generateSnapshotHeader(name, packageInfo, out);
        if (out.checkError()) {
            throw new IOException();
        }
        BinaryHprofWriter.write(samplingProfiler.getHprofData(), outputStream);
    } catch (IOException e) {
        Log.e(TAG, "Error writing snapshot to " + path, e);
        return;
    } finally {
        IoUtils.closeQuietly(outputStream);
    }
    new File(path).setReadable(true, false);

    long elapsed = System.currentTimeMillis() - start;
    Log.i(TAG, "Wrote snapshot " + path + " in " + elapsed + "ms.");
    samplingProfiler.start(samplingProfilerMilliseconds);
}
```

SamplingProfilerIntegration 的核心是类 SamplingProfiler，这个类在如下文件中定义：

```
libcore/dalvik/src/main/java/dalvik/system/profiler/SamplingProfiler.java
```

文件 SamplingProfiler.java 的具体实现代码如下所示：

```
public final class SamplingProfiler {
    private final Map<HprofData.StackTrace, int[]> stackTraces =
            new HashMap<HprofData.StackTrace, int[]>();
```

```
private final HprofData hprofData = new HprofData(stackTraces);
private final Timer timer = new Timer("SamplingProfiler", true);
private Sampler sampler;
private final int depth;
private final ThreadSet threadSet;
private int nextThreadId = 200001;
private int nextStackTraceId = 300001;
private int nextObjectId = 1;
private Thread[] currentThreads = new Thread[0];
private final Map<Thread, Integer> threadIds = new HashMap<Thread, Integer>();
private final HprofData.StackTrace mutableStackTrace = new HprofData.StackTrace();
private final ThreadSampler threadSampler;
public SamplingProfiler(int depth, ThreadSet threadSet) {
    this.depth = depth;
    this.threadSet = threadSet;
    this.threadSampler = findDefaultThreadSampler();
    threadSampler.setDepth(depth);
    hprofData.setFlags(BinaryHprof.ControlSettings.CPU_SAMPLING.bitmask);
    hprofData.setDepth(depth);
}

private static ThreadSampler findDefaultThreadSampler() {
    if ("Dalvik Core Library".equals(
      System.getProperty("java.specification.name"))) {
       String className = "dalvik.system.profiler.DalvikThreadSampler";
       try {
           return (ThreadSampler)Class.forName(className).newInstance();
       } catch (Exception e) {
           System.out.println("Problem creating " + className + ": " + e);
       }
    }
    return new PortableThreadSampler();
}
public static interface ThreadSet {
    public Thread[] threads();
}
public static ThreadSet newArrayThreadSet(Thread... threads) {
    return new ArrayThreadSet(threads);
}
private static class ArrayThreadSet implements ThreadSet {
    private final Thread[] threads;
    public ArrayThreadSet(Thread... threads) {
        if (threads == null) {
            throw new NullPointerException("threads == null");
        }
        this.threads = threads;
    }
    public Thread[] threads() {
        return threads;
```

```
        }
    }
    public static ThreadSet newThreadGroupTheadSet(ThreadGroup threadGroup) {
        return new ThreadGroupThreadSet(threadGroup);
    }

    private static class ThreadGroupThreadSet implements ThreadSet {
        private final ThreadGroup threadGroup;
        private Thread[] threads;
        private int lastThread;

        public ThreadGroupThreadSet(ThreadGroup threadGroup) {
            if (threadGroup == null) {
                throw new NullPointerException("threadGroup == null");
            }
            this.threadGroup = threadGroup;
            resize();
        }

        private void resize() {
            int count = threadGroup.activeCount();
            threads = new Thread[count*2];
            lastThread = 0;
        }

        public Thread[] threads() {
            int threadCount;
            while (true) {
                threadCount = threadGroup.enumerate(threads);
                if (threadCount == threads.length) {
                    resize();
                } else {
                    break;
                }
            }
            if (threadCount < lastThread) {
                // avoid retaining pointers to threads that have ended
                Arrays.fill(threads, threadCount, lastThread, null);
            }
            lastThread = threadCount;
            return threads;
        }
    }

    public void start(int interval) {
        if (interval < 1) {
            throw new IllegalArgumentException("interval < 1");
        }
        if (sampler != null) {
```

```
            throw new IllegalStateException("profiling already started");
        }
        sampler = new Sampler();
        hprofData.setStartMillis(System.currentTimeMillis());
        timer.scheduleAtFixedRate(sampler, 0, interval);
    }

    public void stop() {
        if (sampler == null) {
            return;
        }
        synchronized(sampler) {
            sampler.stop = true;
            while (!sampler.stopped) {
                try {
                    sampler.wait();
                } catch (InterruptedException ignored) {}
            }
        }
        sampler = null;
    }
    public void shutdown() {
        stop();
        timer.cancel();
    }

    public HprofData getHprofData() {
        if (sampler != null) {
            throw new IllegalStateException("cannot access hprof data while sampling");
        }
        return hprofData;
    }

    private class Sampler extends TimerTask {

        private boolean stop;
        private boolean stopped;

        private Thread timerThread;

        public void run() {
            synchronized(this) {
                if (stop) {
                    cancel();
                    stopped = true;
                    notifyAll();
                    return;
                }
            }
```

```
        if (timerThread == null) {
            timerThread = Thread.currentThread();
        }
        Thread[] newThreads = threadSet.threads();
        if (!Arrays.equals(currentThreads, newThreads)) {
            updateThreadHistory(currentThreads, newThreads);
            currentThreads = newThreads.clone();
        }

        for (Thread thread : currentThreads) {
            if (thread == null) {
                break;
            }
            if (thread == timerThread) {
                continue;
            }

            StackTraceElement[] stackFrames = threadSampler.getStackTrace(thread);
            if (stackFrames == null) {
                continue;
            }
            recordStackTrace(thread, stackFrames);
        }
    }
    private void recordStackTrace(Thread thread, StackTraceElement[] stackFrames) {
        Integer threadId = threadIds.get(thread);
        if (threadId == null) {
            throw new IllegalArgumentException("Unknown thread " + thread);
        }
        mutableStackTrace.threadId = threadId;
        mutableStackTrace.stackFrames = stackFrames;

        int[] countCell = stackTraces.get(mutableStackTrace);
        if (countCell == null) {
            countCell = new int[1];
            // cloned because the ThreadSampler may reuse the array
            StackTraceElement[] stackFramesCopy = stackFrames.clone();
            HprofData.StackTrace stackTrace =
                    new HprofData.StackTrace(
                        nextStackTraceId++, threadId, stackFramesCopy);
            hprofData.addStackTrace(stackTrace, countCell);
        }
        countCell[0]++;
    }

    private void updateThreadHistory(Thread[] oldThreads, Thread[] newThreads) {
        Set<Thread> n = new HashSet<Thread>(Arrays.asList(newThreads));
        Set<Thread> o = new HashSet<Thread>(Arrays.asList(oldThreads));
```

```
        Set<Thread> added = new HashSet<Thread>(n);
        added.removeAll(o);

        Set<Thread> removed = new HashSet<Thread>(o);
        removed.removeAll(n);

        for (Thread thread : added) {
            if (thread == null) {
                continue;
            }
            if (thread == timerThread) {
                continue;
            }
            addStartThread(thread);
        }
        for (Thread thread : removed) {
            if (thread == null) {
                continue;
            }
            if (thread == timerThread) {
                continue;
            }
            addEndThread(thread);
        }
    }

    private void addStartThread(Thread thread) {
        if (thread == null) {
            throw new NullPointerException("thread == null");
        }
        int threadId = nextThreadId++;
        Integer old = threadIds.put(thread, threadId);
        if (old != null) {
            throw new IllegalArgumentException("Thread already registered as " + old);
        }

        String threadName = thread.getName();
        // group will become null when thread is terminated
        ThreadGroup group = thread.getThreadGroup();
        String groupName = group == null ? null : group.getName();
        ThreadGroup parentGroup = group == null ? null : group.getParent();
        String parentGroupName = parentGroup == null ? null : parentGroup.getName();

        HprofData.ThreadEvent event =
                HprofData.ThreadEvent.start(nextObjectId++, threadId,
                                threadName, groupName, parentGroupName);
        hprofData.addThreadEvent(event);
    }
```

```
    private void addEndThread(Thread thread) {
        if (thread == null) {
            throw new NullPointerException("thread == null");
        }
        Integer threadId = threadIds.remove(thread);
        if (threadId == null) {
            throw new IllegalArgumentException("Unknown thread " + thread);
        }
        HprofData.ThreadEvent event = HprofData.ThreadEvent.end(threadId);
        hprofData.addThreadEvent(event);
    }
  }
}
```

7.3 应用程序进程详解

在启动 Android 应用程序的过程中，除了可以获得虚拟机实例外，还可以获得一个消息循环和一个 Binder 线程池。这样，在应用程序中运行的组件可以使用系统的信息处理机制和 Binder 通信机制实现自己的业务逻辑。本节将详细分析创建应用程序的实现源码，为读者步入本书后面知识的学习打下基础。

7.3.1 创建应用程序

在 Android 系统中，当 ActivityManagerService 创建新进程来启动某个应用程序组件时，会调用类 ActivityManagerService 中的函数 startProcessLocked 向孵化进程 Zygote 发送创建应用程序进程的请求。函数 startProcessLocked 在文件 frameworks\base\services\java\com\android\server\am\ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final void startProcessLocked(ProcessRecord app, String hostingType,
  String hostingNameStr) {
    if (app.pid>0 && app.pid!=MY_PID) {
        synchronized(mPidsSelfLocked) {
            mPidsSelfLocked.remove(app.pid);
            mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);
        }
        app.setPid(0);
    }

    if (DEBUG_PROCESSES && mProcessesOnHold.contains(app))
        Slog.v(TAG, "startProcessLocked removing on hold: " + app);
    mProcessesOnHold.remove(app);

    updateCpuStats();
```

```
System.arraycopy(mProcDeaths, 0, mProcDeaths, 1, mProcDeaths.length-1);
mProcDeaths[0] = 0;
//获取创建应用程序进程的用户 ID 和用户组 ID
try {
    int uid = app.uid;

    int[] gids = null;
    int mountExternal = Zygote.MOUNT_EXTERNAL_NONE;
    if (!app.isolated) {
        int[] permGids = null;
        try {
            final PackageManager pm = mContext.getPackageManager();
            permGids = pm.getPackageGids(app.info.packageName);

            if (Environment.isExternalStorageEmulated()) {
                if (pm.checkPermission(
                  android.Manifest.permission.ACCESS_ALL_EXTERNAL_STORAGE,
                  app.info.packageName) == PERMISSION_GRANTED) {
                    mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL;
                } else {
                    mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER;
                }
            }
        } catch (PackageManager.NameNotFoundException e) {
            Slog.w(TAG, "Unable to retrieve gids", e);
        }

        if (permGids == null) {
            gids = new int[1];
        } else {
            gids = new int[permGids.length + 1];
            System.arraycopy(permGids, 0, gids, 1, permGids.length);
        }
        gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppId(uid));
    }
    if (mFactoryTest != SystemServer.FACTORY_TEST_OFF) {
        if (mFactoryTest==SystemServer.FACTORY_TEST_LOW_LEVEL
          && mTopComponent!=null
          && app.processName.equals(mTopComponent.getPackageName())) {
            uid = 0;
        }
        if (mFactoryTest==SystemServer.FACTORY_TEST_HIGH_LEVEL
          && (app.info.flags & ApplicationInfo.FLAG_FACTORY_TEST)!=0) {
            uid = 0;
        }
    }
    int debugFlags = 0;
    if ((app.info.flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
        debugFlags |= Zygote.DEBUG_ENABLE_DEBUGGER;
```

```
        // Also turn on CheckJNI for debuggable apps. It's quite
        // awkward to turn on otherwise.
        debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
    }
    if ((app.info.flags & ApplicationInfo.FLAG_VM_SAFE_MODE)!=0
      || Zygote.systemInSafeMode == true) {
        debugFlags |= Zygote.DEBUG_ENABLE_SAFEMODE;
    }
    if ("1".equals(SystemProperties.get("debug.checkjni"))) {
        debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
    }
    if ("1".equals(SystemProperties.get("debug.jni.logging"))) {
        debugFlags |= Zygote.DEBUG_ENABLE_JNI_LOGGING;
    }
    if ("1".equals(SystemProperties.get("debug.assert"))) {
        debugFlags |= Zygote.DEBUG_ENABLE_ASSERT;
    }
    //调用函数 start 创建应用程序进程
    Process.ProcessStartResult startResult =
      Process.start("android.app.ActivityThread",
        app.processName, uid, uid, gids, debugFlags, mountExternal,
        app.info.targetSdkVersion, app.info.seinfo, null);
    BatteryStatsImpl bs = app.batteryStats.getBatteryStats();
    synchronized (bs) {
        if (bs.isOnBattery()) {
            app.batteryStats.incStartsLocked();
        }
    }

    EventLog.writeEvent(EventLogTags.AM_PROC_START,
      UserHandle.getUserId(uid), startResult.pid, uid,
      app.processName, hostingType,
      hostingNameStr != null ? hostingNameStr : "");
    if (app.persistent) {
        Watchdog.getInstance().processStarted(
          app.processName, startResult.pid);
    }
    StringBuilder buf = mStringBuilder;
    buf.setLength(0);
    buf.append("Start proc ");
    buf.append(app.processName);
    buf.append(" for ");
    buf.append(hostingType);
    if (hostingNameStr != null) {
        buf.append(" ");
        buf.append(hostingNameStr);
    }
    buf.append(": pid=");
    buf.append(startResult.pid);
```

```
            buf.append(" uid=");
            buf.append(uid);
            buf.append(" gids={");
            if (gids != null) {
                for (int gi=0; gi<gids.length; gi++) {
                    if (gi != 0) buf.append(", ");
                    buf.append(gids[gi]);
                }
            }
            buf.append("}");
            Slog.i(TAG, buf.toString());
            app.setPid(startResult.pid);
            app.usingWrapper = startResult.usingWrapper;
            app.removed = false;
            synchronized (mPidsSelfLocked) {
                this.mPidsSelfLocked.put(startResult.pid, app);
                Message msg = mHandler.obtainMessage(PROC_START_TIMEOUT_MSG);
                msg.obj = app;
                mHandler.sendMessageDelayed(msg, startResult.usingWrapper ?
                  PROC_START_TIMEOUT_WITH_WRAPPER : PROC_START_TIMEOUT);
            }
        } catch (RuntimeException e) {
            // XXX do better error recovery.
            app.setPid(0);
            Slog.e(TAG, "Failure starting process " + app.processName, e);
        }
}
```

类 Process 中的函数 start 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
public static final ProcessStartResult start(final String processClass,
  final String niceName, int uid, int gid, int[] gids, int debugFlags,
  int mountExternal, int targetSdkVersion, String seInfo, String[] zygoteArgs) {
    try {
        //调用函数 startViaZygote 让 Zygote 进程创建一个应用程序进程
        return startViaZygote(processClass, niceName, uid, gid, gids,
          debugFlags, mountExternal, targetSdkVersion, seInfo, zygoteArgs);
    } catch (ZygoteStartFailedEx ex) {
        Log.e(LOG_TAG, "Starting VM process through Zygote failed");
        throw new RuntimeException(
          "Starting VM process through Zygote failed", ex);
    }
}
```

在上述代码中，用到了函数 startViaZygote，功能是将要创建的应用程序进程的启动参数保存在字符串列表 argsForZygote 中，并调用函数 zygoteSendArgsAndGetResult 请求进程 Zygote 创建应用程序。函数 startViaZygote 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
private static ProcessStartResult startViaZygote(final String processClass,
  final String niceName, final int uid, final int gid, final int[] gids,
  int debugFlags, int mountExternal, int targetSdkVersion, String seInfo,
  String[] extraArgs) throws ZygoteStartFailedEx {
    synchronized(Process.class) {
    ArrayList<String> argsForZygote = new ArrayList<String>();
    // --runtime-init, --setuid=, --setgid=,
    // and --setgroups= must go first
    argsForZygote.add("--runtime-init");
    argsForZygote.add("--setuid=" + uid);
    argsForZygote.add("--setgid=" + gid);
    if ((debugFlags & Zygote.DEBUG_ENABLE_JNI_LOGGING) != 0) {
        argsForZygote.add("--enable-jni-logging");
    }
    if ((debugFlags & Zygote.DEBUG_ENABLE_SAFEMODE) != 0) {
        argsForZygote.add("--enable-safemode");
    }
    if ((debugFlags & Zygote.DEBUG_ENABLE_DEBUGGER) != 0) {
        argsForZygote.add("--enable-debugger");
    }
    if ((debugFlags & Zygote.DEBUG_ENABLE_CHECKJNI) != 0) {
        argsForZygote.add("--enable-checkjni");
    }
    if ((debugFlags & Zygote.DEBUG_ENABLE_ASSERT) != 0) {
        argsForZygote.add("--enable-assert");
    }
    if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER) {
        argsForZygote.add("--mount-external-multiuser");
    } else if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL) {
        argsForZygote.add("--mount-external-multiuser-all");
    }
    argsForZygote.add("--target-sdk-version=" + targetSdkVersion);
    //TODO optionally enable debuger
    //argsForZygote.add("--enable-debugger");
    // --setgroups is a comma-separated list
    if (gids!=null && gids.length>0) {
        StringBuilder sb = new StringBuilder();
        sb.append("--setgroups=");
        int sz = gids.length;
        for (int i=0; i<sz; i++) {
            if (i != 0) {
                sb.append(',');
            }
            sb.append(gids[i]);
        }
        argsForZygote.add(sb.toString());
    }
    if (niceName != null) {
```

```
        argsForZygote.add("--nice-name=" + niceName);
    }
    if (seInfo != null) {
        argsForZygote.add("--seinfo=" + seInfo);
    }
    argsForZygote.add(processClass);
    if (extraArgs != null) {
        for (String arg : extraArgs) {
            argsForZygote.add(arg);
        }
    }
    //请求进程 Zygote 创建应用程序
    return zygoteSendArgsAndGetResult(argsForZygote);
}
```

在上述代码中，通过函数 zygoteSendArgsAndGetResult 调用 Zygote 进程创建了一个指定的应用程序。函数 zygoteSendArgsAndGetResult 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
private static ProcessStartResult zygoteSendArgsAndGetResult(
  ArrayList<String> args) throws ZygoteStartFailedEx {
    //调用函数 openZygoteSocketIfNeeded 创建一个连接到 Zygote 进程的本地对象 LocalSocket
    openZygoteSocketIfNeeded();
    try {
        /**
        * 将要创建的应用程序进程启动参数列表写入到本地对象 LocalSocket 中
        * Zygote 进程接收到数据之后，会创建一个新的应用程序进程
        *将创建的进程 pid 返回给 ActivityManagerService
        */
        sZygoteWriter.write(Integer.toString(args.size()));
        sZygoteWriter.newLine();
        int sz = args.size();
        for (int i=0; i<sz; i++) {
            String arg = args.get(i);
            if (arg.indexOf('\n') >= 0) {
                throw new ZygoteStartFailedEx("embedded newlines not allowed");
            }
            sZygoteWriter.write(arg);
            sZygoteWriter.newLine();
        }
        sZygoteWriter.flush();
        // Should there be a timeout on this?
        ProcessStartResult result = new ProcessStartResult();
        result.pid = sZygoteInputStream.readInt();
        if (result.pid < 0) {
            throw new ZygoteStartFailedEx("fork() failed");
        }
        result.usingWrapper = sZygoteInputStream.readBoolean();
        return result;
```

```
    } catch (IOException ex) {
        try {
            if (sZygoteSocket != null) {
                sZygoteSocket.close();
            }
        } catch (IOException ex2) {
            // we're going to fail anyway
            Log.e(LOG_TAG, "I/O exception on routine close", ex2);
        }
        sZygoteSocket = null;
        throw new ZygoteStartFailedEx(ex);
    }
}
```

在上述代码中，用到了函数 openZygoteSocketIfNeeded，功能是创建一个连接到 Zygote 进程的本地对象 LocalSocket。函数 openZygoteSocketIfNeeded 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
private static void openZygoteSocketIfNeeded() throws ZygoteStartFailedEx {
    int retryCount;
    if (sPreviousZygoteOpenFailed) {
        /*
        * If we've failed before, expect that we'll fail again and
        * don't pause for retries.
        */
        retryCount = 0;
    } else {
        retryCount = 10;
    }
    for (int retry=0; (sZygoteSocket==null)&&(retry<(retryCount + 1));
      retry++) {
        if (retry > 0) {
            try {
                Log.i("Zygote", "Zygote not up yet, sleeping...");
                Thread.sleep(ZYGOTE_RETRY_MILLIS);
            } catch (InterruptedException ex) {
                // should never happen
            }
        }
        try {
            //创建一个保存在 sZygoteSocket 中的 LocalSocket 对象
            sZygoteSocket = new LocalSocket();
            //将创建的的 LocalSocket 对象与名为 ZYGOTE_SOCKET 的 zygote 进程建立连接
            sZygoteSocket.connect(new LocalSocketAddress(
              ZYGOTE_SOCKET, LocalSocketAddress.Namespace.RESERVED));
            //将获得的 LocalSocket 对象 sZygoteSocket 的输入流保存在
            //变量 sZygoteInputStream 中
            sZygoteInputStream =
              new DataInputStream(sZygoteSocket.getInputStream());
```

```
            //将获得的 LocalSocket 对象 sZygoteSocket 的输出流保存在变量 sZygoteWriter 中
            sZygoteWriter = new BufferedWriter(
              new OutputStreamWriter(sZygoteSocket.getOutputStream()), 256);
            Log.i("Zygote", "Process: zygote socket opened");
            sPreviousZygoteOpenFailed = false;
            break;
        } catch (IOException ex) {
            if (sZygoteSocket != null) {
                try {
                    sZygoteSocket.close();
                } catch (IOException ex2) {
                    Log.e(LOG_TAG, "I/O exception on close after exception", ex2);
                }
            }
            sZygoteSocket = null;
        }
    }
    if (sZygoteSocket == null) {
        sPreviousZygoteOpenFailed = true;
        throw new ZygoteStartFailedEx("connect failed");
    }
}
```

在上述代码中，sZygoteSocket 是一个 LocalSocket 类型的成员变量，能够连接 Zygote 进程中的名为“zygote”的 Socket，这个 Socket 与设备文件/dev/socket/zygote 相对应。

接下来，Zygote 进程会在函数 runSelectLoop 中接收一个创建新应用程序的要求。

函数 runSelectLoop 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void runSelectLoop() throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);

    int loopCount = GC_LOOP_COUNT;
    while (true) {
        int index;
        if (loopCount <= 0) {
            gc();
            loopCount = GC_LOOP_COUNT;
        } else {
            loopCount--;
        }

        try {
            fdArray = fds.toArray(fdArray);
```

```
            index = selectReadable(fdArray);
        } catch (IOException ex) {
            throw new RuntimeException("Error in select()", ex);
        }

        if (index < 0) {
            throw new RuntimeException("Error in select()");
        } else if (index == 0) {
            ZygoteConnection newPeer = acceptCommandPeer();
            peers.add(newPeer);
            fds.add(newPeer.getFileDesciptor());
        } else {
            boolean done;
            done = peers.get(index).runOnce();
            if (done) {
                peers.remove(index);
                fds.remove(index);
            }
        }
    }
}
```

在上述代码中，会调用函数 runOnce 处理接收到的创建新应用程序的要求。函数 runOnce 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java 中定义，具体实现代码如下所示：

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {

    String args[];
    Arguments parsedArgs = null;
    FileDescriptor[] descriptors;

    try {
        args = readArgumentList(); //获得启动要创建应用程序进程的参数
        descriptors = mSocket.getAncillaryFileDescriptors();
    } catch (IOException ex) {
        Log.w(TAG, "IOException on command socket " + ex.getMessage());
        closeSocket();
        return true;
    }

    if (args == null) {
        // EOF reached.
        closeSocket();
        return true;
    }

    /** the stderr of the most recent request, if avail */
    PrintStream newStderr = null;
```

```
if (descriptors!=null && descriptors.length>=3) {
    newStderr = new PrintStream(new FileOutputStream(descriptors[2]));
}

int pid = -1;
FileDescriptor childPipeFd = null;
FileDescriptor serverPipeFd = null;

try {
    parsedArgs = new Arguments(args);

    applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContext);
    applyRlimitSecurityPolicy(parsedArgs, peer, peerSecurityContext);
    applyCapabilitiesSecurityPolicy(parsedArgs, peer, peerSecurityContext);
    applyInvokeWithSecurityPolicy(parsedArgs, peer, peerSecurityContext);
    applyseInfoSecurityPolicy(parsedArgs, peer, peerSecurityContext);

    applyDebuggerSystemProperty(parsedArgs);
    applyInvokeWithSystemProperty(parsedArgs);

    int[][] rlimits = null;

    if (parsedArgs.rlimits != null) {
        rlimits = parsedArgs.rlimits.toArray(intArray2d);
    }

    if (parsedArgs.runtimeInit && parsedArgs.invokeWith!=null) {
        FileDescriptor[] pipeFds = Libcore.os.pipe();
        childPipeFd = pipeFds[1];
        serverPipeFd = pipeFds[0];
        ZygoteInit.setCloseOnExec(serverPipeFd, true);
    }
    //调用函数 forkAndSpecialize 创建应用程序进程
    pid = Zygote.forkAndSpecialize(
      parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
      parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal,
      parsedArgs.seInfo, parsedArgs.niceName);
} catch (IOException ex) {
    logAndPrintError(newStderr, "Exception creating pipe", ex);
} catch (ErrnoException ex) {
    logAndPrintError(newStderr, "Exception creating pipe", ex);
} catch (IllegalArgumentException ex) {
    logAndPrintError(newStderr, "Invalid zygote arguments", ex);
} catch (ZygoteSecurityException ex) {
    logAndPrintError(
      newStderr, "Zygote security policy prevents request: ", ex);
}
```

```
    try {
        if (pid == 0) {
            // in child
            IoUtils.closeQuietly(serverPipeFd);
            serverPipeFd = null;
            handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);

            // should never get here, the child is expected to either
            // throw ZygoteInit.MethodAndArgsCaller or exec().
            return true;
        } else {
            // in parent...pid of < 0 means failure
            IoUtils.closeQuietly(childPipeFd);
            childPipeFd = null;
            return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
        }
    } finally {
        IoUtils.closeQuietly(childPipeFd);
        IoUtils.closeQuietly(serverPipeFd);
    }
}
```

在上述代码中，通过函数 readArgumentList 获得启动要创建应用程序进程的参数，并通过函数 forkAndSpecialize 创建了这个要启动应用程序的进程。其中函数 readArgumentList 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java 中定义，具体实现代码如下所示：

```
private String[] readArgumentList() throws IOException {

    /**
    * See android.os.Process.zygoteSendArgsAndGetPid()
    * Presently the wire format to the zygote process is:
    * a) a count of arguments (argc, in essence)
    * b) a number of newline-separated argument strings equal to count
    *
    * After the zygote process reads these it will write the pid of
    * the child or -1 on failure.
    */
    int argc;
    try {
        String s = mSocketReader.readLine();
        if (s == null) {
            // EOF reached.
            return null;
        }
        argc = Integer.parseInt(s);
    } catch (NumberFormatException ex) {
        Log.e(TAG, "invalid Zygote wire format: non-int at argc");
        throw new IOException("invalid wire format");
```

```
    }

    if (argc > MAX_ZYGOTE_ARGC) {
        throw new IOException("max arg count exceeded");
    }

    String[] result = new String[argc];
    for (int i=0; i<argc; i++) {
        result[i] = mSocketReader.readLine();
        if (result[i] == null) {
            // We got an unexpected EOF.
            throw new IOException("truncated request");
        }
    }
    return result;
}
```

函数 forkAndSpecialize 在文件 libcore\dalvik\src\main\java\dalvik\system\Zygote.java 中定义，具体实现代码如下所示：

```
public static int forkAndSpecialize(int uid, int gid, int[] gids, int debugFlags,
  int[][] rlimits, int mountExternal, String seInfo, String niceName) {
    preFork();
    int pid = nativeForkAndSpecialize(
      uid, gid, gids, debugFlags, rlimits, mountExternal, seInfo, niceName);
    postFork();
    return pid;
}
```

在上述代码中，当创建一个进程的子进程时，如果返回值为 0，则表示在新创建的进程中执行。此时需要调用函数 handleChildProc 来启动这个子进程，并在 handleChildProc 中调用函数 zygoteInit 在新创建的应用程序进程中初始化运行库，这样便可以启动一个 Binder 线程池。

7.3.2　启动线程池

在创建新应用程序完毕之前，需要调用类 RuntimeInit 中的函数 nativeZygoteInit 启动一个新的 Binder 线程池，具体启动流程如下所示。

调用类 RuntimeInit 中的函数 nativeZygoteInit，此函数在文件 frameworks\base\core\java\com\android\internal\os\RuntimeInit.java 中定义，对应的代码如下所示：

```
public class RuntimeInit {
    private final static String TAG = "AndroidRuntime";
    private final static boolean DEBUG = false;

    private static boolean initialized;

    private static IBinder mApplicationObject;
```

```
    private static volatile boolean mCrashing = false;

    private static final native void nativeZygoteInit();
    private static final native void nativeFinishInit();
```

函数 nativeZygoteInit 是一个 JNI 函数，在文件 frameworks\base\core\jni\AndroidRuntime.cpp 中定义实现，对应的代码如下所示：

```
static void com android internal os RuntimeInit nativeZygoteInit(
  JNIEnv *env, jobject clazz)
{
    gCurRuntime->onZygoteInit();
}
```

在上述实现代码中，gCurRuntime 是一个全局变量，上述代码用到了 gCurRuntime 的成员函数 onZygoteInit，启动了一个 Binder 线程池。

函数 onZygoteInit 在文件 frameworks\base\cmds\app_process\app_main.cpp 中定义，具体实现代码如下所示：

```
virtual void onZygoteInit() {
    // Re-enable tracing now that we're no longer in Zygote.
    atrace_set_tracing_enabled(true);

    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    //调用函数 startThreadPool 启动一个 Binder 线程池
    proc->startThreadPool();
}
```

在上述代码中，当调用函数 startThreadPool 启动一个 Binder 线程池后，当前应用程序进程就可以通过 Binder 机制与其他进程实现通信了。

函数 startThreadPool 在文件 frameworks\native\libs\binder\ProcessState.cpp 中定义，具体实现代码如下所示：

```
void ProcessState::startThreadPool() {
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true; //默认值为 false
        spawnPooledThread(true);
    }
}
```

在上述代码中，mThreadPoolStarted 的默认值为 false。当第一次调用函数 startThreadPool 时，会在当前进程中启动 Binder 线程池，并将 mThreadPoolStarted 设置为 true，这样做的目的是防止在以后重复启动 Binder 线程池。

7.3.3 创建信息循环

当创建新应用程序进程完毕以后，会调用函数 invokeStaticMain，将类 ActivityThread 的函

数 main 设置为新程序的入口函数。当使用函数 main 时，会在当前程序的进程中建立一个信息循环。

接下来首先看函数 invokeStaticMain 的具体实现，此函数在文件 frameworks\base\core\java\com\android\internal\os\RuntimeInit.java 中定义，具体实现代码如下所示：

```
private static void invokeStaticMain(String className, String[] argv)
  throws ZygoteInit.MethodAndArgsCaller {
    Class<?> cl;

    try {
        cl = Class.forName(className);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
        "Missing class when invoking static main " + className, ex);
    }

    Method m;
    try {
        //获得静态成员函数 main，并保存在 Method 对象中
        m = cl.getMethod("main", new Class[] { String[].class });
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException("Missing static main on " + className, ex);
    } catch (SecurityException ex) {
        throw new RuntimeException(
          "Problem getting static main on " + className, ex);
    }

    int modifiers = m.getModifiers();
    if (!(Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
        throw new RuntimeException(
          "Main method is not public and static on " + className);
    }

    /*
    *将 method 对象封装在静态成员函数 main 中，并保存在一个 Method 对象中
    * 将 MethodAndArgsCaller 对象作为异常抛给当前程序进程来处理
    */
    throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

静态成员函数 main 在 frameworks\base\core\java\com\android\internal\os\ RuntimeInit.java 文件中定义，具体实现代码如下所示：

```
public static final void main(String[] argv) {
    if (argv.length==2 && argv[1].equals("application")) {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application");
        redirectLogStreams();
    } else {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting tool");
```

```
    }

    commonInit();
    nativeFinishInit();

    if (DEBUG) Slog.d(TAG, "Leaving RuntimeInit!");
}
```

上述代码中，函数 main 捕获到 MethodAndArgsCaller 异常后，会调用 MethodAndArgsCaller 成员函数 run 进行后面的处理。接下来看函数 MethodAndArgsCaller 和 run，这两个函数都是在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义的，具体实现代码如下所示：

```
public static class MethodAndArgsCaller extends Exception
  implements Runnable {
    /** method to call */
    private final Method mMethod;

    /** argument array */
    private final String[] mArgs;

    public MethodAndArgsCaller(Method method, String[] args) {
        mMethod = method;
        mArgs = args;
    }

    public void run() {
        try {
            //执行函数 invoke，这样就执行了类 android.app.ActivityThread 中的函数 main
            mMethod.invoke(null, new Object[] { mArgs });
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(ex);
        } catch (InvocationTargetException ex) {
            Throwable cause = ex.getCause();
            if (cause instanceof RuntimeException) {
                throw (RuntimeException)cause;
            } else if (cause instanceof Error) {
                throw (Error)cause;
            }
            throw new RuntimeException(ex);
        }
    }
}
```

在上述代码中，变量 mMethod 和 mArgs 是在构造异常对象时传递进来的，其中变量 mMethod 与类 android.app.ActivityThread 中的函数 main 相对应。

Android

第 8 章
分析Activity组件

在 Android 系统中，Activity、Service、Broadcast 和 ContentProvider 是最重要的核心组件。在本书前面的分析内存系统源码的内容中，曾经介绍过 Service 组件的一些知识。本章的内容中，将详细分析 Android 4.3 系统中 Activity 组件的源码，为读者步入本书后面知识的学习打下基础。

8.1 Activity基础

在 Android 程序中，Activity 通常的表现形式是一个单独的界面(screen)。每个 Activity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。

8.1.1 Activity的状态

在 Android 系统中，大多数应用程序会有多个 Activity。例如，一个文本信息程序会有如下所示的界面：

- 显示联系人列表的界面。
- 写信息界面。
- 查看信息界面或设置界面。

上述每个界面都是一个 Activity，切换到另一个界面就是载入一个新的 Activity。在某些情况下，一个 Activity 可能会给前一个 Activity 返回值，例如，一个让用户选择相片的 Activity 会把选择到的相片返回给其调用者。当打开一个新界面后，前一个界面就被暂停，并放入历史栈中(界面切换历史栈)。使用者可以回溯前面已经打开的存放在历史栈中的界面，也可以从历史栈中删除没有界面价值的界面。Android 在历史栈中保留程序运行产生的所有界面：从第一个界面到最后一个。

当 Activity 被创建或销毁时，它们进入或退出 Activity 栈。当它们做这些动作时，就会在如下 4 种可能的状态间迁移。

- Active：当 Activity 在栈的顶端时，它是可见的、有焦点的前台 Activity，用来响应用户的输入。Android 会不惜一切代价来尝试保证它的活跃性，需要的话，它会消灭栈中更靠下的 Activity，以保证 Active Activity 需要的资源。当另一个 Activity 变成 Active 状态时，这个 Activity 就会变成 Paused。
- Paused：在一些情况下，你的 Activity 可见，但不拥有焦点；在这个时刻，它就是暂停的。当最前面的 Activity 是全透明或非全屏的 Activity 时，下面的 Activity 就会到达这个状态。当暂停时，这个 Activity 还是被看作是 Active 的，但不接受用户的输入事件。在极端的情况下，Android 会杀死一个 Paused 的 Activity 来恢复资源给 Active Activity。当一个 Activity 完全不可见时，它就变成 Stopped。
- Stopped：当一个 Activity 不可见时，它就“停止”了。这个 Activity 仍然留在内存里来保存所有的状态和成员信息；但是，在什么地方当系统需要内存时，它就是“罪犯”，拉出去枪毙了。当一个 Activity 停止时，保存数据和当前 UI 状态是很重要的。一旦 Activity 退出或关闭，它就变成 Inactive 状态。
- Inactive：当一个曾经被启动过的 Activity 被杀死时，它就变成 Inactive。Inactive Activity 会从 Activity 栈中移除，当它重新显示和使用时，需要再次启动。

Activity 的状态转换关系如图 8-1 所示。

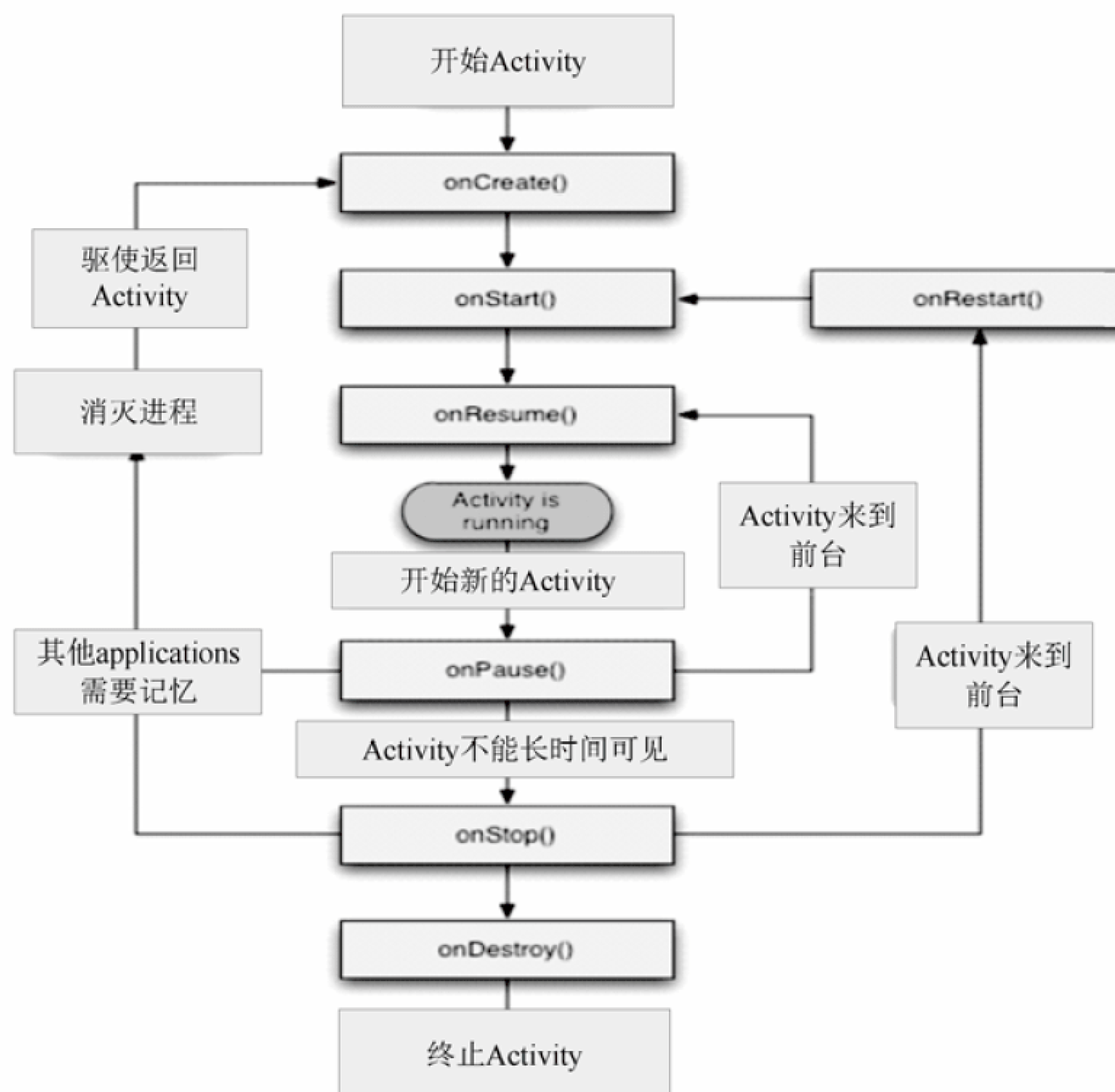


图 8-1　Activity的状态转换关系

状态的变化是人为的，完全由 Android 内存管理器掌握着。Android 会首先关闭那些包含 Inactive Activity 的应用程序，其次关闭那些 Stopped 的程序，在极端情况时会移除那些 Paused 的程序。

为了保证无瑕疵的用户体验，这些状态的迁移对用户来说必须是不可见的。当 Activity 从 Paused、Stopped 或者杀死的状态返回到 Active 的时候，UI 必须是无差别的。所以，当 Activity 暂停或停止时，保存所有的 UI 状态和数据是很重要的。一旦 Activity 变成 Active，它需要从保存的值中恢复。

8.1.2　Activity的主要函数

(1)　void onCreate(Bundle savedInstanceState)

该函数在 Activity 被第一次加载时执行。我们新启动一个程序的时候，其主窗体的 onCreate 事件就会被执行。如果 Activity 被销毁后(onDestroy 后)，再重新加载进 Task 时，其 onCreate 事件也会被重新执行。注意这里的参数 savedInstanceState(Bundle 类型是一个键值对集合，可以看成是.NET 中的 Dictionary)是一个很有用的设计，由于前面已经说到过的手机应用的特殊性，一个 Activity 很可能被强制交换到后台(交换到后台就是指该窗体不再对用户可见，但实际上又还是存在于某个 Task 中的，比如一个新的 Activity 压入了当前的 Task，从而“遮盖”住了当前的 Activity，或者用户按了 Home 键回到桌面，又或者其他重要事件发生，导致新的 Activity 出现在当前 Activity 之上，比如来电界面)，而如果此后用户在一段时间内没有重新查

看该窗体(Android 通过长按 Home 键可以选择最近运行的 6 个程序，或者用户直接再次点击程序的运行图标，如果窗体所在的 Task 和进程没有被系统销毁，则不用重新加载 Process、Task 和 Task 中的 Activity，直接重新显示 Task 顶部的 Activity，这就称为重新查看某个程序的窗体)，该窗体连同其所在的 Task 和 Process 则可能已经被系统自动销毁了，此时如果再次查看该窗体，则要重新执行 onCreate 事件初始化窗体。而这个时候，我们可能希望用户继续从上次打开该窗体时的操作状态进行操作，而不是一切从头开始。例如用户在编辑短信时突然来电，接完电话后用户又去做了一些其他的事情，比如保存来电号码到联系人，而没有立即回到短信编辑界面，导致了短信编辑界面被销毁，当用户重新进入短信程序时，他可能希望继续上次的编辑。此时就可以覆写 Activity 的 void onSaveInstanceState(Bundle outState)事件，通过向 outState 中写入一些我们需要在窗体销毁前保存的状态或信息，这样在窗体重新执行 onCreate 的时候，就会通过 savedInstanceState 将先前保存的信息传递进来，此时我们就可以有选择地利用这些信息来初始化窗体了，而不是一切从头开始。

(2) void onStart()

该函数在 onCreate 事件之后执行；或者当前窗体被交换到后台后，在用户重新查看窗体前已经过去了一段时间，窗体已经执行了 onStop 事件，但是窗体和其所在进程并没有被销毁，用户再次重新查看窗体时，会执行 onRestart 事件，之后会跳过 onCreate 事件，直接执行窗体的 onStart 事件。

(3) void onResume()

该函数在 onStart 事件之后执行；或者当前窗体被交换到后台后，在用户重新查看窗体时，窗体还没有被销毁，也没有执行过 onStop 事件(窗体还继续存在于 Task 中)，则会跳过窗体的 onCreate 和 onStart 事件，直接执行 onResume 事件。

(4) void onPause()

该函数在窗体被交换到后台时执行。

(5) void onStop()

该函数在 onPause 事件之后执行。如果一段时间内用户还没有重新查看该窗体，则该窗体的 onStop 事件将会被执行；或者用户直接按了 Back 键，将该窗体从当前 Task 中移除，也会执行该窗体的 onStop 事件。

(6) void onRestart()

onStop 事件执行后，如果窗体和其所在的进程没有被系统销毁，此时用户又重新查看该窗体，则会执行窗体的 onRestart 事件，onRestart 事件后，会跳过窗体的 onCreate 事件，直接执行 onStart 事件。

(7) void onDestroy()

该函数在 Activity 被销毁的时候执行。在窗体的 onStop 事件之后，如果没有再次查看该窗体，Activity 会被销毁。

最后，我们用一个实际的例子来说明 Activity 的各个生命周期。假设有一个程序由两个 Activity A 和 B 组成，A 是这个程序的启动界面。当用户启动程序时，Process 和默认的 Task 分别被创建，接着 A 被压入到当前的 Task 中，依次执行了 onCreate、onStart、onResume 事件，被呈现给了用户；此时用户选择 A 中的某个功能开启界面 B，界面 B 被压入当前 Task，遮盖住了 A，A 的 onPause 事件执行，B 的 onCreate、onStart、onResume 事件执行，呈现了界面 B

给用户；用户在界面 B 操作完成后，使用 Back 键回到界面 A，界面 B 不再可见，界面 B 的 onPause、onStop、onDestroy 执行，A 的 onResume 事件被执行，呈现界面 A 给用户。此时突然来电，界面 A 的 onPause 事件被执行，电话接听界面被呈现给用户，用户接听完电话后，又按了 Home 键回到桌面，打开另一个程序“联系人”，添加了联系人信息，又做了一些其他的操作，此时界面 A 不再可见，其 onStop 事件被执行，但并没有被销毁。此后，用户重新从菜单中点击了我们的程序，由于 A 和其所在的进程和 Task 并没有被销毁，A 的 onRestart 和 onStart 事件被执行，接着 A 的 onResume 事件被执行，A 又被呈现给了用户。用户这次使用完后，按 Back 键返回到桌面，A 的 onPause、onStop 被执行，随后 A 的 onDestroy 被执行，由于当前 Task 中已经没有任何 Activity，A 所在的 Process 的重要程度被降到很低，很快 A 所在的 Process 被系统结束。

8.2　启动Activity

在 Android 系统的应用程序框架层中，ActivityManagerService 负责启动 Activity。在整个启动过程中，ActivityManagerService 用于管理 Activity 的生命周期，ActivityManagerService 可以通过 ActivityStack 将所有的 Activity 按照后进先出的顺序放在一个堆栈中。对于每一个应用程序来说，都拥有一个专门的 ActivityThread 来表示应用程序的主进程。在每个 ActivityThread 中，都包含了一个 Binder 对象类型的 ApplicationThread 实例，其功能是与其他进程实现通信。

具体地说，在 Android 系统中启动 Activity 的流程如下所示。

(1) 通过 Binder 进程间通信进入到 ActivityManagerService 进程中，并且将会调用 ActivityManagerService.startActivity 接口。

(2) ActivityManagerService 调用 ActivityStack.startActivityMayWait，做好启动 Activity 前的准备。

(3) ActivityStack 通知 ApplicationThread 即将启动 Activity，ApplicationThread 表示调用 ActivityManagerService.startActivity 接口的进程。

(4) ApplicationThread 并不执行真正的启动操作，它通过调用 ActivityManagerService.activityPaused 接口进入到 ActivityManagerService 进程中，查看是否需要创建新的进程来启动 Activity。

(5) 对于通过点击应用程序图标来启动 Activity 的情形来说，ActivityManagerService 会在本步调用 startProcessLocked 创建一个新的进程。而对于通过在 Activity 内部调用 startActivity 来启动新的 Activity 来说，并不需要执行这一步，因为新的 Activity 就在原来的 Activity 所在的进程中进行启动。

(6) ActivityManagerServic 调用 ApplicationThread.scheduleLaunchActivity 接口，通知相应的进程执行启动 Activity 的操作。

(7) ApplicationThread 把这个启动 Activity 的操作转发给 ActivityThread，ActivityThread 通过 ClassLoader 导入相应的 Activity 类，然后把它启动起来。

在本节的内容中，将以根 Activity 组件中的 MainActivity 为例，讲解启动 MainActivity 的具体流程。

8.2.1 Launcher启动应用程序

在 Android 系统中，Launcher 负责启动应用程序。当在 Android 中安装应用程序后，会在 Launcher 界面出现一个相应的图标，点击这个图标后，Launcher 会启动这个图标对应的应用程序。其实 Launcher 也是一个应用程序，在 Android 4.3 的源码中，Launcher 应用程序的源码被保存在\packages\apps\Launcher2 目录中。

在文件 packages\apps\Launcher2\src\com\android\launcher2\Launcher.java 中，用于启动 Android 应用程序的实现源码如下所示：

```
public void onClick(View v) {
    // Make sure that rogue clicks don't get through while allapps is launching,
    // or after the view has detached (it's possible for this to happen
    // if the view is removed mid touch).
    if (v.getWindowToken() == null) {
        return;
    }

    if (!mWorkspace.isFinishedSwitchingState()) {
        return;
    }

    Object tag = v.getTag();
    if (tag instanceof ShortcutInfo) {
        // Open shortcut
        final Intent intent = ((ShortcutInfo)tag).intent;
        int[] pos = new int[2];
        v.getLocationOnScreen(pos);
        intent.setSourceBounds(new Rect(pos[0], pos[1],
          pos[0] + v.getWidth(), pos[1] + v.getHeight()));

        boolean success = startActivitySafely(v, intent, tag);

        if (success && v instanceof BubbleTextView) {
            mWaitingForResume = (BubbleTextView)v;
            mWaitingForResume.setStayPressed(true);
        }
    } else if (tag instanceof FolderInfo) {
        if (v instanceof FolderIcon) {
            FolderIcon fi = (FolderIcon)v;
            handleFolderClick(fi);
        }
    } else if (v == mAllAppsButton) {
        if (isAllAppsVisible()) {
            showWorkspace(true);
        } else {
            onClickAllAppsButton(v);
        }
```

```
    }
}
boolean startActivitySafely(View v, Intent intent, Object tag) {
    boolean success = false;
    try {
        success = startActivity(v, intent, tag);
    } catch (ActivityNotFoundException e) {
        Toast.makeText(
          this, R.string.activity_not_found, Toast.LENGTH_SHORT).show();
        Log.e(TAG, "Unable to launch. tag=" + tag + " intent=" + intent, e);
    }
    return success;
}
```

另外，在类 Activity 中定义了函数 startActivity 的具体实现，此函数的功能是调用函数 startActivityForResult 进一步实现启动应用程序处理。

函数 startActivity 在文件 frameworks/base/core/java/android/app/Activity.java 中定义，具体实现代码如下所示：

```
public void startActivity(Intent intent) {
    startActivity(intent, null);
}
```

在上述代码中，用到了函数 startActivity，其第 2 个参数传入 null，表示不需要这个 Activity 结束后的返回结果。函数 startActivity 在文件 frameworks/base/core/java/android/app/Activity.java 中定义，具体实现代码如下所示：

```
public void startActivityForResult(
  Intent intent, int requestCode, Bundle options) {
    if (mParent == null) {
        Instrumentation.ActivityResult ar = mInstrumentation.execStartActivity(
          this, mMainThread.getApplicationThread(), mToken, this,
          intent, requestCode, options);
        if (ar != null) {
            mMainThread.sendActivityResult(
              mToken, mEmbeddedID, requestCode, ar.getResultCode(),
              ar.getResultData());
        }
        if (requestCode >= 0) {
            // If this start is requesting a result, we can avoid making
            // the activity visible until the result is received.  Setting
            // this code during onCreate(Bundle savedInstanceState)
            // or onResume() will keep the
            // activity hidden during this time, to avoid flickering.
            // This can only be done when a result is requested because
            // that guarantees we will get information back when the
            // activity is finished, no matter what happens to it.
            mStartedActivity = true;
        }
```

```
    } else {
        if (options != null) {
            mParent.startActivityFromChild(this, intent, requestCode, options);
        } else {
            // Note we want to go through this method for compatibility with
            // existing applications that may have overridden it.
            mParent.startActivityFromChild(this, intent, requestCode);
        }
    }
}
```

在上述代码中，mInstrumentation 是类 Activity 中类型为 Instrumentation 的成员，在文件 frameworks/base/core/java/android/app/Instrumentation.java 中定义，功能是监控应用程序与系统的交互。

8.2.2 返回ActivityManagerService的远程接口

再看函数 execStartActivity，该函数的功能是返回 ActivityManagerService 的远程接口，即 ActivityManagerProxy 接口。

函数 execStartActivity 在文件 frameworks/base/core/java/android/app/Instrumentation.java 中定义，具体实现代码如下所示：

```
public ActivityResult execStartActivity(
  Context who, IBinder contextThread, IBinder token, Activity target,
  Intent intent, int requestCode, Bundle options) {
    IApplicationThread whoThread = (IApplicationThread)contextThread;
    if (mActivityMonitors != null) {
        synchronized (mSync) {
            final int N = mActivityMonitors.size();
            for (int i=0; i<N; i++) {
                final ActivityMonitor am = mActivityMonitors.get(i);
                if (am.match(who, null, intent)) {
                    am.mHits++;
                    if (am.isBlocking()) {
                        return requestCode >= 0? am.getResult() : null;
                    }
                    break;
                }
            }
        }
    }
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess();
        int result = ActivityManagerNative.getDefault().startActivity(
          whoThread, who.getBasePackageName(), intent,
          intent.resolveTypeIfNeeded(who.getContentResolver()),
          token, target!=null? target.mEmbeddedID : null,
```

```
            requestCode, 0, null, null, options);
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {}
    return null;
}
```

再看类 ActivityManagerService 中的函数 startActivity，功能是将我们的操作转发给成员变量 mMainStack 的 startActivityMayWait 函数，此处 mMainStack 的类型为 ActivityStack。函数 startActivity 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public final int startActivity(IApplicationThread caller, String callingPackage,
  Intent intent, String resolvedType, IBinder resultTo,
  String resultWho, int requestCode, int startFlags,
  String profileFile, ParcelFileDescriptor profileFd, Bundle options) {
    return startActivityAsUser(
      caller, callingPackage, intent, resolvedType, resultTo,
      resultWho, requestCode, startFlags, profileFile, profileFd,
      options, UserHandle.getCallingUserId());
}
```

8.2.3　解析intent的内容

再看函数 startActivityMayWait，如果前面步骤中的参数 outResult 和 config 都是 null，并且如下表达式的结果为 false：

```
(aInfo.applicationInfo.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0
```

则解析参数 intent 的内容，并将得到的 MainActivity 信息保存在变量 aInfo 中。

函数 startActivityMayWait 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final int startActivityMayWait(IApplicationThread caller, int callingUid,
  String callingPackage, Intent intent, String resolvedType, IBinder resultTo,
  String resultWho, int requestCode, int startFlags, String profileFile,
  ParcelFileDescriptor profileFd, WaitResult outResult, Configuration config,
  Bundle options, int userId) {
    // Refuse possible leaked file descriptors
    if (intent!=null && intent.hasFileDescriptors()) {
        throw new IllegalArgumentException("File descriptors passed in Intent");
    }
    boolean componentSpecified = intent.getComponent()!=null;

    // Don't modify the client's object!
    intent = new Intent(intent);

    // Collect information about the target of the Intent.
    ActivityInfo aInfo = resolveActivity(
```

```
        intent, resolvedType, startFlags, profileFile, profileFd, userId);

    synchronized (mService) {
        int callingPid;
        if (callingUid >= 0) {
            callingPid = -1;
        } else if (caller == null) {
            callingPid = Binder.getCallingPid();
            callingUid = Binder.getCallingUid();
        } else {
            callingPid = callingUid = -1;
        }

        mConfigWillChange =
          config!=null && mService.mConfiguration.diff(config) != 0;
        if (DEBUG_CONFIGURATION)
            Slog.v(TAG, "Starting activity when config will change = "
              + mConfigWillChange);

        final long origId = Binder.clearCallingIdentity();

        if (mMainStack && aInfo!=null
          && (aInfo.applicationInfo.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE)
          != 0) {
            // This may be a heavy-weight process!  Check to see if we already
            // have another, different heavy-weight process running.
            if (aInfo.processName.equals(aInfo.applicationInfo.packageName)) {
                if (mService.mHeavyWeightProcess!=null
                  && (mService.mHeavyWeightProcess.info.uid
                  != aInfo.applicationInfo.uid
                  || !mService.mHeavyWeightProcess.processName
                  .equals(aInfo.processName))) {
                    int realCallingPid = callingPid;
                    int realCallingUid = callingUid;
                    if (caller != null) {
                        ProcessRecord callerApp =
                          mService.getRecordForAppLocked(caller);
                        if (callerApp != null) {
                            realCallingPid = callerApp.pid;
                            realCallingUid = callerApp.info.uid;
                        } else {
                            Slog.w(TAG, "Unable to find app for caller " + caller
                              + " (pid=" + realCallingPid + ") when starting: "
                              + intent.toString());
                            ActivityOptions.abort(options);
                            return ActivityManager.START_PERMISSION_DENIED;
                        }
                    }
```

```
    IIntentSender target = mService.getIntentSenderLocked(
      ActivityManager.INTENT SENDER ACTIVITY, "android",
      realCallingUid, userId, null, null, 0, new Intent[] { intent },
      new String[] { resolvedType },
      PendingIntent.FLAG_CANCEL_CURRENT
      | PendingIntent.FLAG_ONE_SHOT,
      null);

    Intent newIntent = new Intent();
    if (requestCode >= 0) {
        // Caller is requesting a result.
        newIntent.putExtra(
          HeavyWeightSwitcherActivity.KEY_HAS_RESULT, true);
    }
    newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_INTENT,
      new IntentSender(target));
    if (mService.mHeavyWeightProcess.activities.size() > 0) {
        ActivityRecord hist =
          mService.mHeavyWeightProcess.activities.get(0);
        newIntent.putExtra(
          HeavyWeightSwitcherActivity.KEY  CUR APP,
          hist.packageName);
        newIntent.putExtra(
          HeavyWeightSwitcherActivity.KEY_CUR_TASK,
          hist.task.taskId);
    }
    newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_ NEW_APP,
      aInfo.packageName);
    newIntent.setFlags(intent.getFlags());
    newIntent.setClassName("android",
      HeavyWeightSwitcherActivity.class.getName());
    intent = newIntent;
    resolvedType = null;
    caller = null;
    callingUid = Binder.getCallingUid();
    callingPid = Binder.getCallingPid();
    componentSpecified = true;
    try {
        ResolveInfo rInfo =
          AppGlobals.getPackageManager().resolveIntent(
            intent, null,
            PackageManager.MATCH_DEFAULT_ONLY
            | ActivityManagerService.STOCK_PM_FLAGS, userId);
        aInfo = rInfo!=null ? rInfo.activityInfo : null;
        aInfo = mService.getActivityInfoForUser(aInfo, userId);
    } catch (RemoteException e) {
        aInfo = null;
    }
}
```

```
        }
    }

    int res = startActivityLocked(caller, intent, resolvedType,
      aInfo, resultTo, resultWho, requestCode, callingPid, callingUid,
      callingPackage, startFlags, options, componentSpecified, null);

    if (mConfigWillChange && mMainStack) {
        // If the caller also wants to switch to a new configuration,
        // do so now.  This allows a clean switch, as we are waiting
        // for the current activity to pause (so we will not destroy
        // it), and have not yet started the next activity.
        mService.enforceCallingPermission(
          android.Manifest.permission.CHANGE_CONFIGURATION,
          "updateConfiguration()");
        mConfigWillChange = false;
        if (DEBUG_CONFIGURATION)
            Slog.v(TAG,
                "Updating to new configuration after starting activity.");
        mService.updateConfigurationLocked(config, null, false, false);
    }

    Binder.restoreCallingIdentity(origId);

    if (outResult != null) {
        outResult.result = res;
        if (res == ActivityManager.START_SUCCESS) {
            mWaitingActivityLaunched.add(outResult);
            do {
                try {
                    mService.wait();
                } catch (InterruptedException e) {}
            } while (!outResult.timeout && outResult.who==null);
        } else if (res == ActivityManager.START_TASK_TO_FRONT) {
            ActivityRecord r = this.topRunningActivityLocked(null);
            if (r.nowVisible) {
                outResult.timeout = false;
                outResult.who =
                  new ComponentName(r.info.packageName, r.info.name);
                outResult.totalTime = 0;
                outResult.thisTime = 0;
            } else {
                outResult.thisTime = SystemClock.uptimeMillis();
                mWaitingActivityVisible.add(outResult);
                do {
                    try {
                        mService.wait();
                    } catch (InterruptedException e) {}
                } while (!outResult.timeout && outResult.who == null);
```

```
            }
        }
    }
    return res;
  }
}
```

8.2.4　分析检查机制

再看函数 startActivityLocked，后缀 Locked 表示这个函数是线程安全的，即该函数体只能由一个线程调用。如果另一个线程也需要调用该函数，必须等待前一个线程执行完毕。函数 startActivityLocked 的主要功能如下所示：

- 检查当前正在运行的 Activity 是否与目标 Activity 一致，如果一致，则直接返回。也就是说，程序员在 Activity 使用 startActivity 时启动自己，不会达到重启当前 Activity 的目的。
- 处理 INTENT_FLAG_FORWARD_RESULT 标志，在代码中，此标志能够跨 Activity 传递 Result。
- 检查是否存在 Caller 的 app，此功能发生在发出 startActivity 命令后，Caller 所在的进程被意外杀死，此时 AmS 拒绝继续往下执行。
- 检查 Caller 是否具备启动指定 Activity 的权限。

函数 startActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final int startActivityLocked(IApplicationThread caller, Intent intent,
  String resolvedType, ActivityInfo aInfo, IBinder resultTo,
  String resultWho, int requestCode, int callingPid, int callingUid,
  String callingPackage, int startFlags, Bundle options,
  boolean componentSpecified, ActivityRecord[] outActivity) {

    int err = ActivityManager.START_SUCCESS;

    ProcessRecord callerApp = null;

    if (caller != null) {
        callerApp = mService.getRecordForAppLocked(caller);
        if (callerApp != null) {
            callingPid = callerApp.pid;
            callingUid = callerApp.info.uid;
        } else {
            Slog.w(TAG, "Unable to find app for caller " + caller
              + " (pid=" + callingPid + ") when starting: "
              + intent.toString());
            err = ActivityManager.START_PERMISSION_DENIED;
        }
    }
```

```
if (err == ActivityManager.START SUCCESS) {
    final int userId = aInfo!=null?
      UserHandle.getUserId(aInfo.applicationInfo.uid) : 0;
    Slog.i(TAG, "START u" + userId + " {"
      + intent.toShortString(true, true, true, false)
      + "} from pid " + (callerApp != null ? callerApp.pid : callingPid));
}

ActivityRecord sourceRecord = null;
ActivityRecord resultRecord = null;
if (resultTo != null) {
    int index = indexOfTokenLocked(resultTo);
    if (DEBUG_RESULTS) Slog.v(
      TAG, "Will send result to " + resultTo + " (index " + index + ")");
    if (index >= 0) {
        sourceRecord = mHistory.get(index);
        if (requestCode >= 0 && !sourceRecord.finishing) {
            resultRecord = sourceRecord;
        }
    }
}

int launchFlags = intent.getFlags();

if ((launchFlags&Intent.FLAG_ACTIVITY_FORWARD_RESULT)!=0
  && sourceRecord != null) {
    // Transfer the result target from the source activity to the new
    // one being started, including any failures.
    if (requestCode >= 0) {
        ActivityOptions.abort(options);
        return ActivityManager.START_FORWARD_AND_REQUEST_CONFLICT;
    }
    resultRecord = sourceRecord.resultTo;
    resultWho = sourceRecord.resultWho;
    requestCode = sourceRecord.requestCode;
    sourceRecord.resultTo = null;
    if (resultRecord != null) {
        resultRecord.removeResultsLocked(
        sourceRecord, resultWho, requestCode);
    }
}

if (err == ActivityManager.START_SUCCESS && intent.getComponent() == null) {
    // We couldn't find a class that can handle the given Intent.
    // That's the end of that!
    err = ActivityManager.START_INTENT_NOT_RESOLVED;
}
```

```
if (err == ActivityManager.START_SUCCESS && aInfo == null) {
    // We couldn't find the specific class specified in the Intent.
    // Also the end of the line.
    err = ActivityManager.START_CLASS_NOT_FOUND;
}

if (err != ActivityManager.START_SUCCESS) {
    if (resultRecord != null) {
        sendActivityResultLocked(-1,
        resultRecord, resultWho, requestCode,
        Activity.RESULT_CANCELED, null);
    }
    mDismissKeyguardOnNextActivity = false;
    ActivityOptions.abort(options);
    return err;
}

final int startAnyPerm = mService.checkPermission(
  START_ANY_ACTIVITY, callingPid, callingUid);
final int componentPerm = mService.checkComponentPermission(
  aInfo.permission, callingPid, callingUid,
  aInfo.applicationInfo.uid, aInfo.exported);
if (startAnyPerm != PERMISSION_GRANTED
  && componentPerm != PERMISSION_GRANTED) {
    if (resultRecord != null) {
        sendActivityResultLocked(-1,
          resultRecord, resultWho, requestCode,
          Activity.RESULT_CANCELED, null);
    }
    mDismissKeyguardOnNextActivity = false;
    String msg;
    if (!aInfo.exported) {
        msg = "Permission Denial: starting " + intent.toString()
          + " from " + callerApp + " (pid=" + callingPid
          + ", uid=" + callingUid + ")"
          + " not exported from uid " + aInfo.applicationInfo.uid;
    } else {
        msg = "Permission Denial: starting " + intent.toString()
          + " from " + callerApp + " (pid=" + callingPid
          + ", uid=" + callingUid + ")"
          + " requires " + aInfo.permission;
    }
    Slog.w(TAG, msg);
    throw new SecurityException(msg);
}

boolean abort = !mService.mIntentFirewall.checkStartActivity(intent,
  callerApp==null? null : callerApp.info, callingUid,
  callingPid, resolvedType, aInfo);
```

```
if (mMainStack) {
    if (mService.mController != null) {
        try {
            // The Intent we give to the watcher has the extra data
            // stripped off, since it can contain private information.
            Intent watchIntent = intent.cloneFilter();
            abort |= !mService.mController.activityStarting(
              watchIntent, aInfo.applicationInfo.packageName);
        } catch (RemoteException e) {
            mService.mController = null;
        }
    }
}

if (abort) {
    if (resultRecord != null) {
        sendActivityResultLocked(-1,
          resultRecord, resultWho, requestCode,
          Activity.RESULT_CANCELED, null);
    }
    // We pretend to the caller that it was really started, but
    // they will just get a cancel result.
    mDismissKeyguardOnNextActivity = false;
    ActivityOptions.abort(options);
    return ActivityManager.START_SUCCESS;
}
//创建即将要启动的Activity的相关信息，并保存在r变量中
ActivityRecord r = new ActivityRecord(mService, this, callerApp,
  callingUid, callingPackage, intent, resolvedType, aInfo,
  mService.mConfiguration, resultRecord, resultWho,
  requestCode, componentSpecified);
if (outActivity != null) {
    outActivity[0] = r;
}

if (mMainStack) {
    if (mResumedActivity == null
      || mResumedActivity.info.applicationInfo.uid != callingUid) {
        if (!mService.checkAppSwitchAllowedLocked(callingPid, callingUid,
          "Activity start")) {
            PendingActivityLaunch pal = new PendingActivityLaunch();
            pal.r = r;
            pal.sourceRecord = sourceRecord;
            pal.startFlags = startFlags;
            mService.mPendingActivityLaunches.add(pal);
            mDismissKeyguardOnNextActivity = false;
            ActivityOptions.abort(options);
            return ActivityManager.START_SWITCHES_CANCELED;
```

```
            }
        }

        if (mService.mDidAppSwitch) {
            // This is the second allowed switch since we stopped switches,
            // so now just generally allow switches.  Use case: user presses
            // home (switches disabled, switch to home, mDidAppSwitch now true);
            // user taps a home icon (coming from home so allowed, we hit here
            // and now allow anyone to switch again).
            mService.mAppSwitchesAllowedTime = 0;
        } else {
            mService.mDidAppSwitch = true;
        }

        mService.doPendingActivityLaunchesLocked(false);
    }

    err = startActivityUncheckedLocked(r, sourceRecord,
    startFlags, true, options);
    if (mDismissKeyguardOnNextActivity && mPausingActivity == null) {
        // Someone asked to have the keyguard dismissed on the next
        // activity start, but we are not actually doing an activity
        // switch...  just dismiss the keyguard now, because we
        // probably want to see whatever is behind it.
        mDismissKeyguardOnNextActivity = false;
        mService.mWindowManager.dismissKeyguard();
    }
    return err;
}
```

如果等待序列为空，则调用函数 startActivityUncheckedLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，此时要启动的 Activity 已经通过检查机制的检验，并被“诊断”为是一个“健康合法”的启动请求。函数 startActivityUncheckedLocked 的具体实现代码如下所示：

```
final int startActivityUncheckedLocked(ActivityRecord r,
  ActivityRecord sourceRecord, int startFlags, boolean doResume,
  Bundle options) {
    final Intent intent = r.intent;
    final int callingUid = r.launchedFromUid;

    int launchFlags = intent.getFlags();

    // We'll invoke onUserLeaving before onPause only if the launching
    // activity did not explicitly state that this is an automated launch.
    mUserLeaving = (launchFlags&Intent.FLAG_ACTIVITY_NO_USER_ACTION) == 0;
    if (DEBUG_USER_LEAVING)
        Slog.v(TAG, "startActivity() => mUserLeaving=" + mUserLeaving);
```

```
// If the caller has asked not to resume at this point, we make note
// of this in the record so that we can skip it when trying to find
// the top running activity.
if (!doResume) {
    r.delayedResume = true;
}

ActivityRecord notTop =
  (launchFlags&Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP)!=0? r : null;

// If the onlyIfNeeded flag is set, then we can do this if the activity
// being launched is the same as the one making the call... or, as
// a special case, if we do not know the caller then we count the
// current top activity as the caller.
if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED) != 0) {
    ActivityRecord checkedCaller = sourceRecord;
    if (checkedCaller == null) {
        checkedCaller = topRunningNonDelayedActivityLocked(notTop);
    }
    if (!checkedCaller.realActivity.equals(r.realActivity)) {
        // Caller is not the same as launcher, so always needed.
        startFlags &= ~ActivityManager.START_FLAG_ONLY_IF_NEEDED;
    }
}

if (sourceRecord == null) {
    // This activity is not being started from another... in this
    // case we -always- start a new task.
    if ((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
        Slog.w(TAG, "startActivity called from non-Activity context;
          forcing Intent.FLAG_ACTIVITY_NEW_TASK for: "
          + intent);
        launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
    }
} else if (sourceRecord.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
    // The original activity who is starting us is running as a single
    // instance... this new activity it is starting must go on its
    // own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
} else if (r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE
  || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
    // The activity being started is a single instance... it always
    // gets launched into its own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
}

if (r.resultTo != null
  && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
    // For whatever reason this activity is being launched into a new
```

```
        // task...  yet the caller has requested a result back.  Well, that
        // is pretty messed up, so instead immediately send back a cancel
        // and let the new task continue launched as normal without a
        // dependency on its originator.
        Slog.w(TAG,
         "Activity is launching as a new task, so cancelling activity result.");
        sendActivityResultLocked(-1,
          r.resultTo, r.resultWho, r.requestCode,
          Activity.RESULT_CANCELED, null);
        r.resultTo = null;
    }

    boolean addingToTask = false;
    boolean movedHome = false;
    TaskRecord reuseTask = null;
    //如果此 intent 的标志值的位 Intent.FLAG_ACTIVITY_NEW_TASK 被置位，
    //而且 Intent.FLAG_ACTIVITY_MULTIPLE_TASK 没有置位，则执行下面的 if 语句
    //下面代码的功能是查看当前有没有 Task 可以用来执行这个 Activity。
    //因为 r.launchMode 的值不为 ActivityInfo.LAUNCH_SINGLE_INSTANCE，
    //所以它通过函数 findTaskLocked 来查找是否存在这样的 Task，
    //此处返回的结果是 null，即 taskTop 为 null，
    //所以需要创建一个新的 Task 来启动这个 Activity
    if (((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0
      && (launchFlags&Intent.FLAG_ACTIVITY_MULTIPLE_TASK) == 0)
      || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK
      || r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
        // If bring to front is requested, and no result is requested, and
        // we can find a task that was started with this same
        // component, then instead of launching bring that one to the front.
        if (r.resultTo == null) {
            // See if there is a task to bring to the front.  If this is
            // a SINGLE_INSTANCE activity, there can be one and only one
            // instance of it in the history, and it is always in its own
            // unique task, so we do a special search.
            ActivityRecord taskTop =
              r.launchMode != ActivityInfo.LAUNCH_ SINGLE_INSTANCE?
                findTaskLocked(intent, r.info)
                : findActivityLocked(intent, r.info);
            if (taskTop != null) {
                if (taskTop.task.intent == null) {
                    // This task was started because of movement of
                    // the activity based on affinity...  now that we
                    // are actually launching it, we can assign the
                    // base intent.
                    taskTop.task.setIntent(intent, r.info);
                }
                // If the target task is not in the front, then we need
                // to bring it to the front...  except...  well, with
                // SINGLE_TASK_LAUNCH it's not entirely clear.  We'd like
```

```
                // to have the same behavior as if a new instance was
                // being started, which means not bringing it to the front
                // if the caller is not itself in the front.
                ActivityRecord curTop =
                  topRunningNonDelayedActivityLocked(notTop);
                if (curTop != null && curTop.task != taskTop.task) {
                    r.intent.addFlags(Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT);
                    boolean callerAtFront = sourceRecord == null
                      || curTop.task == sourceRecord.task;
                    if (callerAtFront) {
                        // We really do want to push this one into the
                        // user's face, right now.
                        movedHome = true;
                        moveHomeToFrontFromLaunchLocked(launchFlags);
                        moveTaskToFrontLocked(taskTop.task, r, options);
                        options = null;
                    }
                }
                // If the caller has requested that the target task be
                // reset, then do so.
                if ((launchFlags&Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED)
                  != 0) {
                    taskTop = resetTaskIfNeededLocked(taskTop, r);
                }
                if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED)
                  != 0) {
                    // We don't need to start a new activity, and
                    // the client said not to do anything if that
                    // is the case, so this is it!  And for paranoia, make
                    // sure we have correctly resumed the top activity.
                    if (doResume) {
                        resumeTopActivityLocked(null, options);
                    } else {
                        ActivityOptions.abort(options);
                    }
                    return ActivityManager.START_RETURN_INTENT_TO_CALLER;
                }
                if ((launchFlags &
                (Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_ CLEAR_TASK))
                == (Intent.FLAG_ACTIVITY_NEW_TASK|
                Intent.FLAG_ACTIVITY_ CLEAR_TASK)) {
                    // The caller has requested to completely replace any
                    // existing task with its new activity.  Well that should
                    // not be too hard...
                    reuseTask = taskTop.task;
                    performClearTaskLocked(taskTop.task.taskId);
                    reuseTask.setIntent(r.intent, r.info);
                } else if ((launchFlags&Intent.FLAG_ACTIVITY_CLEAR_TOP) != 0
                  || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK
```

```
            || r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
              // In this situation we want to remove all activities
              // from the task up to the one being started.  In most
              // cases this means we are resetting the task to its
              // initial state.
              ActivityRecord top = performClearTaskLocked(
                taskTop.task.taskId, r, launchFlags);
              if (top != null) {
                  if (top.frontOfTask) {
                      // Activity aliases may mean we use different
                      // intents for the top activity, so make sure
                      // the task now has the identity of the new
                      // intent.
                      top.task.setIntent(r.intent, r.info);
                  }
                  logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
                  top.deliverNewIntentLocked(callingUid, r.intent);
              } else {
                  // A special case: we need to
                  // start the activity because it is not currently
                  // running, and the caller has asked to clear the
                  // current task to have this activity at the top.
                  addingToTask = true;
                  // Now pretend like this activity is being started
                  // by the top of its task, so it is put in the
                  // right place.
                  sourceRecord = taskTop;
              }
          } else if (r.realActivity.equals(taskTop.task.realActivity)) {
              // In this case the top activity on the task is the
              // same as the one being launched, so we take that
              // as a request to bring the task to the foreground.
              // If the top activity in the task is the root
              // activity, deliver this new intent to it if it
              // desires.
              if (((launchFlags&Intent.FLAG_ACTIVITY_SINGLE_TOP) != 0
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP)
                && taskTop.realActivity.equals(r.realActivity)) {
                  logStartActivity(
                    EventLogTags.AM_NEW_INTENT, r, taskTop.task);
                  if (taskTop.frontOfTask) {
                      taskTop.task.setIntent(r.intent, r.info);
                  }
                  taskTop.deliverNewIntentLocked(callingUid, r.intent);
              } else if (!r.intent.filterEquals(taskTop.task.intent)) {
                  // In this case we are launching the root activity
                  // of the task, but with a different intent. We
                  // should start a new instance on top.
                  addingToTask = true;
```

```
                        sourceRecord = taskTop;
                    }
                } else if ((launchFlags&Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED)
                  == 0) {
                    // In this case an activity is being launched in to an
                    // existing task, without resetting that task.  This
                    // is typically the situation of launching an activity
                    // from a notification or shortcut.  We want to place
                    // the new activity on top of the current task.
                    addingToTask = true;
                    sourceRecord = taskTop;
                } else if (!taskTop.task.rootWasReset) {
                    // In this case we are launching in to an existing task
                    // that has not yet been started from its front door.
                    // The current task has been brought to the front.
                    // Ideally, we'd probably like to place this new task
                    // at the bottom of its stack, but that's a little hard
                    // to do with the current organization of the code so
                    // for now we'll just drop it.
                    taskTop.task.setIntent(r.intent, r.info);
                }
                if (!addingToTask && reuseTask == null) {
                    // We didn't do anything...  but it was needed (a.k.a., client
                    // don't use that intent!)  And for paranoia, make
                    // sure we have correctly resumed the top activity.
                    if (doResume) {
                        resumeTopActivityLocked(null, options);
                    } else {
                        ActivityOptions.abort(options);
                    }
                    return ActivityManager.START_TASK_TO_FRONT;
                }
            }
        }
    }

    //String uri = r.intent.toURI();
    //Intent intent2 = new Intent(uri);
    //Slog.i(TAG, "Given intent: " + r.intent);
    //Slog.i(TAG, "URI is: " + uri);
    //Slog.i(TAG, "To intent: " + intent2);

    if (r.packageName != null) {
        // 查看当前在堆栈顶端的 Activity 是否就是即将要启动的 Activity,
        // 在有些情况下，如果即将要启动的 Activity 就在堆栈的顶端，
        // 那么就不会重新启动这个 Activity 的另外一个实例
        ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
        if (top!=null && r.resultTo==null) {
            if (top.realActivity.equals(r.realActivity)
```

```
        && top.userId==r.userId) {
          if (top.app!=null && top.app.thread!=null) {
              if ((launchFlags&Intent.FLAG_ACTIVITY_SINGLE_TOP) != 0
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
                  logStartActivity(
                    EventLogTags.AM_NEW_INTENT, top, top.task);
                  // For paranoia, make sure we have correctly
                  // resumed the top activity.
                  if (doResume) {
                      resumeTopActivityLocked(null);
                  }
                  ActivityOptions.abort(options);
                  if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED)
                    != 0) {
                      // We don't need to start a new activity, and
                      // the client said not to do anything if that
                      // is the case, so this is it!
                      return ActivityManager.START_RETURN_INTENT_TO_CALLER;
                  }
                  top.deliverNewIntentLocked(callingUid, r.intent);
                  return ActivityManager.START_DELIVERED_TO_TOP;
              }
          }
        }
    }

} else {
    if (r.resultTo != null) {
        sendActivityResultLocked(-1,
          r.resultTo, r.resultWho, r.requestCode,
          Activity.RESULT_CANCELED, null);
    }
    ActivityOptions.abort(options);
    return ActivityManager.START_CLASS_NOT_FOUND;
}

boolean newTask = false;
boolean keepCurTransition = false;

// 因为要在一个新的 Task 里面来启动这个 Activity 了，所以下面新创建一个 Task
if (r.resultTo == null && !addingToTask
  && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
    if (reuseTask == null) {
        // todo: should do better management of integers.
        mService.mCurTask++;
        if (mService.mCurTask <= 0) {
            mService.mCurTask = 1;
        }
```

```
            r.setTask(
              new TaskRecord(mService.mCurTask, r.info, intent), null, true);
            if (DEBUG_TASKS)
                Slog.v(TAG, "Starting new activity "
                + r + " in new task " + r.task);
        } else {
            r.setTask(reuseTask, reuseTask, true);
        }
        newTask = true;
        if (!movedHome) {
            moveHomeToFrontFromLaunchLocked(launchFlags);
        }

    } else if (sourceRecord != null) {
        if (!addingToTask
          && (launchFlags&Intent.FLAG_ACTIVITY_CLEAR_TOP) != 0) {
            // In this case, we are adding the activity to an existing
            // task, but the caller has asked to clear that task if the
            // activity is already running.
            ActivityRecord top = performClearTaskLocked(
              sourceRecord.task.taskId, r, launchFlags);
            keepCurTransition = true;
            if (top != null) {
                logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
                top.deliverNewIntentLocked(callingUid, r.intent);
                // For paranoia, make sure we have correctly
                // resumed the top activity.
                if (doResume) {
                    resumeTopActivityLocked(null);
                }
                ActivityOptions.abort(options);
                return ActivityManager.START_DELIVERED_TO_TOP;
            }
        } else if (!addingToTask
          && (launchFlags&Intent.FLAG_ACTIVITY_REORDER_TO_FRONT) != 0) {
            // In this case, we are launching an activity in our own task
            // that may already be running somewhere in the history, and
            // we want to shuffle it to the front of the stack if so.
            int where = findActivityInHistoryLocked(r, sourceRecord.task.taskId);
            if (where >= 0) {
                ActivityRecord top = moveActivityToFrontLocked(where);
                logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
                top.updateOptionsLocked(options);
                top.deliverNewIntentLocked(callingUid, r.intent);
                if (doResume) {
                    resumeTopActivityLocked(null);
                }
                return ActivityManager.START_DELIVERED_TO_TOP;
            }
```

```
        }
        // An existing activity is starting this new activity, so we want
        // to keep the new one in the same task as the one that is starting
        // it.
        r.setTask(sourceRecord.task, sourceRecord.thumbHolder, false);
        if (DEBUG_TASKS)
            Slog.v(TAG, "Starting new activity " + r
              + " in existing task " + r.task);

    } else {
        // This not being started from an existing activity, and not part
        // of a new task... just put it in the top task, though these days
        // this case should never happen.
        final int N = mHistory.size();
        ActivityRecord prev = N>0? mHistory.get(N-1) : null;
        r.setTask(prev!=null? prev.task
          : new TaskRecord(mService.mCurTask, r.info, intent), null, true);
        if (DEBUG_TASKS)
            Slog.v(TAG, "Starting new activity " + r
              + " in new guessed " + r.task);
    }

    mService.grantUriPermissionFromIntentLocked(callingUid, r.packageName,
    intent, r.getUriPermissionsLocked());

    if (newTask) {
        EventLog.writeEvent(
          EventLogTags.AM_CREATE_TASK, r.userId, r.task.taskId);
    }
    logStartActivity(EventLogTags.AM_CREATE_ACTIVITY, r, r.task);
    startActivityLocked(r, newTask, doResume, keepCurTransition, options);
    return ActivityManager.START_SUCCESS;
}
```

通过上述代码得到了 intent 的标志值，并保存在了变量 launchFlags 中。

再看函数 startActivityLocked(r, newTask, doResume)，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void startActivityLocked(ActivityRecord r, boolean newTask,
  boolean doResume, boolean keepCurTransition, Bundle options) {
    final int NH = mHistory.size();

    int addPos = -1;

    if (!newTask) {
        // If starting in an existing task, find where that is...
        boolean startIt = true;
        for (int i=NH-1; i>=0; i--) {
            ActivityRecord p = mHistory.get(i);
```

```
            if (p.finishing) {
                continue;
            }
            if (p.task == r.task) {
                // Here it is!  Now, if this is not yet visible to the
                // user, then just add it without starting; it will
                // get started when the user navigates back to it.
                addPos = i+1;
                if (!startIt) {
                    if (DEBUG_ADD_REMOVE) {
                        RuntimeException here = new RuntimeException("here");
                        here.fillInStackTrace();
                        Slog.i(TAG,
                          "Adding activity " + r + " to stack at " + addPos, here);
                    }
                    mHistory.add(addPos, r);
                    r.putInHistory();
                    mService.mWindowManager.addAppToken(addPos, r.appToken,
                      r.task.taskId, r.info.screenOrientation, r.fullscreen,
                     (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN)!=0);
                    if (VALIDATE_TOKENS) {
                        validateAppTokensLocked();
                    }
                    ActivityOptions.abort(options);
                    return;
                }
                break;
            }
            if (p.fullscreen) {
                startIt = false;
            }
        }
    }

    // Place a new activity at top of stack, so it is next to interact
    // with the user.
    if (addPos < 0) {
        addPos = NH;
    }

    // If we are not placing the new activity frontmost, we do not want
    // to deliver the onUserLeaving callback to the actual frontmost
    // activity
    if (addPos < NH) {
        mUserLeaving = false;
        if (DEBUG_USER_LEAVING)
          Slog.v(TAG, "startActivity() behind front, mUserLeaving=false");
    }
```

```
// Slot the activity into the history stack and proceed
if (DEBUG_ADD_REMOVE) {
    RuntimeException here = new RuntimeException("here");
    here.fillInStackTrace();
    Slog.i(TAG, "Adding activity " + r + " to stack at " + addPos, here);
}
mHistory.add(addPos, r);
r.putInHistory();
r.frontOfTask = newTask;
if (NH > 0) {
    // We want to show the starting preview window if we are
    // switching to a new task, or the next activity's process is
    // not currently running.
    boolean showStartingIcon = newTask;
    ProcessRecord proc = r.app;
    if (proc == null) {
        proc = mService.mProcessNames.get(
          r.processName, r.info.applicationInfo.uid);
    }
    if (proc == null || proc.thread == null) {
        showStartingIcon = true;
    }
    if (DEBUG_TRANSITION)
        Slog.v(TAG, "Prepare open transition: starting " + r);
    if ((r.intent.getFlags()&Intent.FLAG_ACTIVITY_NO_ANIMATION) != 0) {
        mService.mWindowManager.prepareAppTransition(
          AppTransition.TRANSIT_NONE, keepCurTransition);
        mNoAnimActivities.add(r);
    } else {
        mService.mWindowManager.prepareAppTransition(newTask?
          AppTransition.TRANSIT_TASK_OPEN
          : AppTransition.TRANSIT_ACTIVITY_OPEN, keepCurTransition);
        mNoAnimActivities.remove(r);
    }
    r.updateOptionsLocked(options);
    mService.mWindowManager.addAppToken(
      addPos, r.appToken, r.task.taskId,
      r.info.screenOrientation, r.fullscreen,
      (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN)!=0);
    boolean doShow = true;
    if (newTask) {
        // Even though this activity is starting fresh, we still need
        // to reset it to make sure we apply affinities to move any
        // existing activities from other tasks in to it.
        // If the caller has requested that the target task be
        // reset, then do so.
        if ((r.intent.getFlags()
          &Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) != 0) {
            resetTaskIfNeededLocked(r, r);
```

```
                doShow = topRunningNonDelayedActivityLocked(null) == r;
            }
        }
        if (SHOW_APP_STARTING_PREVIEW && doShow) {
            // Figure out if we are transitioning from another activity that is
            // "has the same starting icon" as the next one.  This allows the
            // window manager to keep the previous window it had previously
            // created, if it still had one.
            ActivityRecord prev = mResumedActivity;
            if (prev != null) {
                // We don't want to reuse the previous starting preview if:
                // (1) The current activity is in a different task.
                if (prev.task != r.task) prev = null;
                // (2) The current activity is already displayed.
                else if (prev.nowVisible) prev = null;
            }
            mService.mWindowManager.setAppStartingWindow(
              r.appToken, r.packageName, r.theme,
              mService.compatibilityInfoForPackageLocked(
              r.info.applicationInfo), r.nonLocalizedLabel,
              r.labelRes, r.icon, r.windowFlags,
              prev!=null? prev.appToken : null, showStartingIcon);
        }
    } else {
        // If this is the first activity, don't do any fancy animations,
        // because there is nothing for it to animate on top of.
        mService.mWindowManager.addAppToken(addPos, r.appToken, r.task.taskId,
        r.info.screenOrientation, r.fullscreen,
        (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0);
        ActivityOptions.abort(options);
    }
    if (VALIDATE_TOKENS) {
        validateAppTokensLocked();
    }

    if (doResume) {
        resumeTopActivityLocked(null);
    }
}
```

在上述代码中，NH 表示当前系统中历史任务的个数，因为 Launcher 已经跑起来了，所以此处 NH 的值大于 0。并且当参数 doResume 为 true 时，调用函数 resumeTopActivityLocked 实现进一步的操作。

8.2.5 执行Activity组件的操作

在类 ActivityStack 中，当函数 startActivityLocked 通知 WindowManagerService 服务准备好一个 Activity 组件切换操作后，如果参数 doResume 的值为 true，则会继续调用另外一个函数

resumeTopActivityLocked，以继续执行启动参数 r 所描述的一个 Activity 组件的操作。函数 resumeTopActivityLocked 的核心功能是继续执行启动参数 r 所描述的一个 Activity 组件的操作，此函数的实现代码如下所示：

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    ActivityRecord next = topRunningActivityLocked(null);
    final boolean userLeaving = mUserLeaving;
    mUserLeaving = false;
    if (next == null) {
        // There are no more activities!  Let's just start up the
        // Launcher...
        if (mMainStack) {
            ActivityOptions.abort(options);
            return mService.startHomeActivityLocked(mCurrentUser);
        }
    }

    next.delayedResume = false;

    //看要启动的 Activity 是否就是当前处理 Resumed 状态的 Activity，
    //如果是的话，那就什么都不用做，
    //直接返回就可以了；否则再看一下系统当前是否为休眠状态，
    //如果是，则再看看要启动的 Activity 是否就是当前处于堆栈顶端的 Activity，
    //如果是的话，则什么都不用做。
    //如果上面两个条件都不满足，则继续往下执行
    if (mResumedActivity==next && next.state==ActivityState.RESUMED) {

        mService.mWindowManager.executeAppTransition();
        mNoAnimActivities.clear();
        ActivityOptions.abort(options);
        return false;
    }
    if ((mService.mSleeping || mService.mShuttingDown)
      && mLastPausedActivity == next
      && (next.state == ActivityState.PAUSED
      || next.state == ActivityState.STOPPED
      || next.state == ActivityState.STOPPING)) {
        mService.mWindowManager.executeAppTransition();
        mNoAnimActivities.clear();
        ActivityOptions.abort(options);
        return false;
    }
    if (mService.mStartedUsers.get(next.userId) == null) {
        Slog.w(TAG, "Skipping resume of top activity " + next
          + ": user " + next.userId + " is stopped");
        return false;
    }
    mStoppingActivities.remove(next);
```

```
mGoingToSleepActivities.remove(next);
next.sleeping = false;
mWaitingVisibleActivities.remove(next);

next.updateOptionsLocked(options);
if (DEBUG_SWITCH) Slog.v(TAG, "Resuming " + next);
// If we are currently pausing an activity, then don't do anything
// until that is done.
if (mPausingActivity != null) {
    if (DEBUG_SWITCH || DEBUG_PAUSE)
      Slog.v(TAG, "Skip resume: pausing=" + mPausingActivity);
    return false;
}
if (false) {
    if (mLastStartedActivity != null && !mLastStartedActivity.finishing) {
        long now = SystemClock.uptimeMillis();
        final boolean inTime = mLastStartedActivity.startTime!=0
          && (mLastStartedActivity.startTime+START_WARN_TIME)>=now;
        final int lastUid = mLastStartedActivity.info.applicationInfo.uid;
        final int nextUid = next.info.applicationInfo.uid;
        if (inTime && lastUid != nextUid
          && lastUid != next.launchedFromUid
          && mService.checkPermission(
          android.Manifest.permission.STOP_APP_SWITCHES,
          -1, next.launchedFromUid)
          != PackageManager.PERMISSION_GRANTED) {
            mService.showLaunchWarningLocked(mLastStartedActivity, next);
        } else {
            next.startTime = now;
            mLastStartedActivity = next;
        }
    } else {
        next.startTime = SystemClock.uptimeMillis();
        mLastStartedActivity = next;
    }
}

if (mResumedActivity != null) {
    if (DEBUG_SWITCH) Slog.v(TAG, "Skip resume: need to start pausing");
    if (next.app!=null && next.app.thread!=null) {
        mService.updateLruProcessLocked(next.app, false);
    }
    startPausingLocked(userLeaving, false);
    return true;
}
final ActivityRecord last = mLastPausedActivity;
if (mService.mSleeping && last!=null && !last.finishing) {
    if ((last.intent.getFlags()&Intent.FLAG_ACTIVITY_NO_HISTORY)!=0
      || (last.info.flags&ActivityInfo.FLAG_NO_HISTORY) != 0) {
```

```
            if (DEBUG_STATES) {
                Slog.d(TAG, "no-history finish of " + last + " on new resume");
            }
            requestFinishActivityLocked(
              last.appToken, Activity.RESULT_CANCELED, null, "no-history", false);
        }
    }

    if (prev!=null && prev!=next) {
        if (!prev.waitingVisible && next!=null && !next.nowVisible) {
            prev.waitingVisible = true;
            mWaitingVisibleActivities.add(prev);
            if (DEBUG_SWITCH)
                Slog.v(TAG, "Resuming top, waiting visible to hide: " + prev);
        } else {
            if (prev.finishing) {
                mService.mWindowManager.setAppVisibility(prev.appToken, false);
                if (DEBUG_SWITCH)
                    Slog.v(TAG, "Not waiting for visible to hide: "
                      + prev + ", waitingVisible="
                      + (prev != null ? prev.waitingVisible : null)
                      + ", nowVisible=" + next.nowVisible);
            } else {
                if (DEBUG_SWITCH)
                  Slog.v(TAG,
                    "Previous already visible but still waiting to hide: "
                    + prev + ", waitingVisible="
                    + (prev != null ? prev.waitingVisible : null)
                    + ", nowVisible=" + next.nowVisible);
            }
        }
    }
    try {
        AppGlobals.getPackageManager()
          .setPackageStoppedState(next.packageName,
          false, next.userId); /* TODO: Verify if correct userid */
    } catch (RemoteException e1) {
        //
    } catch (IllegalArgumentException e) {
        Slog.w(TAG, "Failed trying to unstop package "
          + next.packageName + ": " + e);
    }
    boolean noAnim = false;
    if (prev != null) {
        if (prev.finishing) {
            if (DEBUG_TRANSITION)
                Slog.v(TAG, "Prepare close transition: prev=" + prev);
            if (mNoAnimActivities.contains(prev)) {
                mService.mWindowManager.prepareAppTransition(
```

```
                AppTransition.TRANSIT_NONE, false);
            } else {
                mService.mWindowManager.prepareAppTransition(
                  prev.task==next.task?
                  AppTransition.TRANSIT_ACTIVITY_CLOSE
                  : AppTransition.TRANSIT_TASK_CLOSE, false);
            }
            mService.mWindowManager.setAppWillBeHidden(prev.appToken);
            mService.mWindowManager.setAppVisibility(prev.appToken, false);
        } else {
            if (DEBUG_TRANSITION)
                Slog.v(TAG, "Prepare open transition: prev=" + prev);
            if (mNoAnimActivities.contains(next)) {
                noAnim = true;
                mService.mWindowManager.prepareAppTransition(
                  AppTransition.TRANSIT_NONE, false);
            } else {
                mService.mWindowManager.prepareAppTransition(
                  prev.task == next.task?
                  AppTransition.TRANSIT_ACTIVITY_OPEN
                  : AppTransition.TRANSIT TASK OPEN, false);
            }
        }
        if (false) {
            mService.mWindowManager.setAppWillBeHidden(prev.appToken);
            mService.mWindowManager.setAppVisibility(prev.appToken, false);
        }
    } else if (mHistory.size() > 1) {
        if (DEBUG TRANSITION)
            Slog.v(TAG, "Prepare open transition: no previous");
        if (mNoAnimActivities.contains(next)) {
            noAnim = true;
            mService.mWindowManager.prepareAppTransition(
              AppTransition.TRANSIT_NONE, false);
        } else {
            mService.mWindowManager.prepareAppTransition(
              AppTransition.TRANSIT_ACTIVITY_OPEN, false);
        }
    }
    if (!noAnim) {
        next.applyOptionsLocked();
    } else {
        next.clearOptionsLocked();
    }

    if (next.app!=null && next.app.thread!=null) {
        if (DEBUG_SWITCH) Slog.v(TAG, "Resume running: " + next);

        // This activity is now becoming visible.
```

```
        mService.mWindowManager.setAppVisibility(next.appToken, true);

        // schedule launch ticks to collect information about slow apps.
        next.startLaunchTickingLocked();

        ActivityRecord lastResumedActivity = mResumedActivity;
        ActivityState lastState = next.state;

        mService.updateCpuStats();

        if (DEBUG_STATES)
            Slog.v(TAG, "Moving to RESUMED: " + next + " (in existing)");
        next.state = ActivityState.RESUMED;
        mResumedActivity = next;
        next.task.touchActiveTime();
        if (mMainStack) {
            mService.addRecentTaskLocked(next.task);
        }
        mService.updateLruProcessLocked(next.app, true);
        updateLRUListLocked(next);

        boolean updated = false;
        if (mMainStack) {
            synchronized (mService) {
                Configuration config =
                  mService.mWindowManager.updateOrientationFromAppTokens(
                  mService.mConfiguration,
                  next.mayFreezeScreenLocked(next.app)? next.appToken : null);
                if (config != null) {
                    next.frozenBeforeDestroy = true;
                }
                updated = mService.updateConfigurationLocked(
                  config, next, false, false);
            }
        }
        if (!updated) {
            ActivityRecord nextNext = topRunningActivityLocked(null);
            if (DEBUG_SWITCH)
                Slog.i(TAG,
                  "Activity config changed during resume: " + next
                   + ", new next: " + nextNext);
            if (nextNext != next) {
                // Do over!
                mHandler.sendEmptyMessage(RESUME_TOP_ACTIVITY_MSG);
            }
            if (mMainStack) {
                mService.setFocusedActivityLocked(next);
            }
            ensureActivitiesVisibleLocked(null, 0);
```

```
            mService.mWindowManager.executeAppTransition();
            mNoAnimActivities.clear();
            return true;
        }

        try {
            // Deliver all pending results.
            ArrayList a = next.results;
            if (a != null) {
                final int N = a.size();
                if (!next.finishing && N>0) {
                    if (DEBUG_RESULTS)
                        Slog.v(TAG, "Delivering results to " + next + ": " + a);
                    next.app.thread.scheduleSendResult(next.appToken, a);
                }
            }

            if (next.newIntents != null) {
                next.app.thread.scheduleNewIntent(
                  next.newIntents, next.appToken);
            }

            EventLog.writeEvent(EventLogTags.AM_RESUME_ACTIVITY,
              next.userId, System.identityHashCode(next),
              next.task.taskId, next.shortComponentName);

            next.sleeping = false;
            showAskCompatModeDialogLocked(next);
            next.app.pendingUiClean = true;
            next.app.thread.scheduleResumeActivity(next.appToken,
              mService.isNextTransitionForward());

            checkReadyForSleepLocked();

        } catch (Exception e) {
            // Whoops, need to restart this activity!
            if (DEBUG_STATES)
                Slog.v(TAG, "Resume failed; resetting state to "
                  + lastState + ": " + next);
            next.state = lastState;
            mResumedActivity = lastResumedActivity;
            Slog.i(TAG, "Restarting because process died: " + next);
            if (!next.hasBeenLaunched) {
                next.hasBeenLaunched = true;
            } else {
                if (SHOW_APP_STARTING_PREVIEW && mMainStack) {
                    mService.mWindowManager.setAppStartingWindow(
                      next.appToken, next.packageName, next.theme,
                    mService.compatibilityInfoForPackageLocked(
```

```
                        next.info.applicationInfo), next.nonLocalizedLabel,
                        next.labelRes, next.icon, next.windowFlags, null, true);
                }
            }
            startSpecificActivityLocked(next, true, false);
            return true;
        }

        try {
            next.visible = true;
            completeResumeLocked(next);
        } catch (Exception e) {
            // If any exception gets thrown, toss away this
            // activity and try the next one.
            Slog.w(TAG, "Exception thrown during resume of " + next, e);
            requestFinishActivityLocked(next.appToken, Activity.RESULT_CANCELED,
              null, "resume-exception", true);
            return true;
        }
        next.stopped = false;

    } else {
        if (!next.hasBeenLaunched) {
            next.hasBeenLaunched = true;
        } else {
            if (SHOW_APP_STARTING_PREVIEW) {
                mService.mWindowManager.setAppStartingWindow(
                  next.appToken, next.packageName, next.theme,
                  mService.compatibilityInfoForPackageLocked(
                    next.info.applicationInfo),
                    next.nonLocalizedLabel,
                    next.labelRes, next.icon, next.windowFlags,
                    null, true);
            }
            if (DEBUG_SWITCH) Slog.v(TAG, "Restarting: " + next);
        }
        startSpecificActivityLocked(next, true, true);
    }

    return true;
}
```

在上述代码中，首先调用函数 topRunningActivityLocked 获得堆栈顶端的 Activity：MainActivity，并保存在变量 next 中，然后把 mUserLeaving 值保存在本地变量 userLeaving 中，并重新设置为 false。此处因为 Launcher 是当前正被执行的 Activity，所以 mResumedActivity 为 Launcher。当处理休眠状态时，mLastPausedActivity 会保存堆栈顶端的 Activity，并且因为当前不是休眠状态，所以 mLastPausedActivity 为 null。

8.2.6 将Launcher推入Paused状态

再看函数 startPausingLocked，功能是将 Launcher 推入 Paused 状态，此函数在文件 frameworks/base/core/java/android/app/ApplicationThreadNative.java 中定义，具体实现代码如下所示：

```
private final void startPausingLocked(boolean userLeaving, boolean uiSleeping) {
    if (mPausingActivity != null) {
        RuntimeException e = new RuntimeException();
        Slog.e(TAG, "Trying to pause when pause is already pending for "
          + mPausingActivity, e);
    }
    ActivityRecord prev = mResumedActivity;
    if (prev == null) {
        RuntimeException e = new RuntimeException();
        Slog.e(TAG, "Trying to pause when nothing is resumed", e);
        resumeTopActivityLocked(null);
        return;
    }
    if (DEBUG_STATES) Slog.v(TAG, "Moving to PAUSING: " + prev);
    else if (DEBUG_PAUSE) Slog.v(TAG, "Start pausing: " + prev);
    mResumedActivity = null;
    mPausingActivity = prev;
    mLastPausedActivity = prev;
    prev.state = ActivityState.PAUSING;
    prev.task.touchActiveTime();
    prev.updateThumbnail(screenshotActivities(prev), null);

    mService.updateCpuStats();

    if (prev.app != null && prev.app.thread != null) {
        if (DEBUG_PAUSE) Slog.v(TAG, "Enqueueing pending pause: " + prev);
        try {
            EventLog.writeEvent(EventLogTags.AM_PAUSE_ACTIVITY,
                    prev.userId, System.identityHashCode(prev),
                    prev.shortComponentName);
            //参数prev.finishing表示prev所代表的Activity是否
            //正在等待结束的Activity列表中，
            //由于Laucher这个Activity还没结束，所以此处为false
            //参数prev.configChangeFlags表示哪些config发生了变化
            prev.app.thread.schedulePauseActivity(prev.appToken, prev.finishing,
              userLeaving, prev.configChangeFlags);
            if (mMainStack) {
                mService.updateUsageStats(prev, false);
            }
        } catch (Exception e) {
            // Ignore exception, if process died other code will cleanup.
```

```
            Slog.w(TAG, "Exception thrown during pause", e);
            mPausingActivity = null;
            mLastPausedActivity = null;
        }
    } else {
        mPausingActivity = null;
        mLastPausedActivity = null;
    }

    if (!mService.mSleeping && !mService.mShuttingDown) {
        mLaunchingActivity.acquire();
        if (!mHandler.hasMessages(LAUNCH_TIMEOUT_MSG)) {
            // To be safe, don't allow the wake lock to be held for too long.
            Message msg = mHandler.obtainMessage(LAUNCH_TIMEOUT_MSG);
            mHandler.sendMessageDelayed(msg, LAUNCH_TIMEOUT);
        }
    }

    if (mPausingActivity != null) {
        if (!uiSleeping) {
            prev.pauseKeyDispatchingLocked();
        } else {
            if (DEBUG_PAUSE) Slog.v(TAG, "Key dispatch not paused for screen off");
        }

        Message msg = mHandler.obtainMessage(PAUSE_TIMEOUT_MSG);
        msg.obj = prev;
        prev.pauseTime = SystemClock.uptimeMillis();
        mHandler.sendMessageDelayed(msg, PAUSE_TIMEOUT);
        if (DEBUG_PAUSE) Slog.v(TAG, "Waiting for pause to complete...");
    } else {
        if (DEBUG_PAUSE) Slog.v(TAG, "Activity not running, resuming next.");
        resumeTopActivityLocked(null);
    }
}
```

在上述代码中，首先将 mResumedActivity 保存在本地变量 prev 中，并取出 Launcher 进程中的 ApplicationThread 对象，通过这个对象来通知 Launcher 此 Activity 即将进入 Paused 状态。

再看函数 schedulePauseActivity，这个函数的功能是通过 Binder 进程间通信机制进入到函数 ApplicationThread.schedulePauseActivity 中。

函数 schedulePauseActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final void schedulePauseActivity(IBinder token, boolean finished,
  boolean userLeaving, int configChanges) {
    queueOrSendMessage(
      //此处 finished 值为 false，因此，
      //queueOrSendMessage 的第一个参数值为 H.PAUSE_ACTIVITY，
```

```
    //表示要暂停 token 所代表的 Activity，即 Launcher
    finished? H.PAUSE_ACTIVITY_FINISHING : H.PAUSE_ACTIVITY, token,
    (userLeaving ? 1 : 0), configChanges);
}
```

在上述代码中，调用了类 ActivityThread 中的成员函数 queueOrSendMessage。

8.2.7　处理消息

再看函数 queueOrSendMessage，功能是先将相关信息组装成一个 msg，然后通过成员变量 mH 发送出去。因为此处 mH 的类型是 H，是一个继承于类 Handler 的 ActivityThread 的内部类，所以最后由类 H.handleMessage 来处理这个消息。

函数 queueOrSendMessage 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义。handleMessage 的主要实现代码如下所示：

```
public final class ActivityThread {
    ...
    private final class H extends Handler {
        ...
        public void handleMessage(Message msg) {
            ...
            switch (msg.what) {
                ...
                case PAUSE_ACTIVITY:
                    handlePauseActivity((IBinder)msg.obj,
                      false, msg.arg1!=0, msg.arg2);
                    maybeSnapshot();
                    break;
                ...
            }
```

在上述代码中，msg.obj 是一个 ActivityRecord 对象的引用，它代表 Launcher 这个 Activity。

再看具体的处理函数 handlePauseActivity，该函数的功能是将 Binder 引用的 token 转换成 ActivityRecord 的远程接口 ActivityClientRecord，然后实现如下所示的三个功能：

- 如果 userLeaving 为 true，则通过调用函数 performUserLeavingActivity 的方式来调用 Activity.onUserLeaveHint，通知 Activity 用户这就要离开它。
- 通过调用函数 performPauseActivity 的方式来调用函数 Activity.onPause，在 Activity 的生命周期中，当要让位于其他的 Activity 时，系统就会调用它本身的 onPause 函数。
- 通知 ActivityManagerService 当前的 Activity 已经进入到 Paused 状态。

函数 handlePauseActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private void handlePauseActivity(IBinder token, boolean finished,
  boolean userLeaving, int configChanges) {
    ActivityClientRecord r = mActivities.get(token);
    if (r != null) {
        if (userLeaving) {
```

```
            performUserLeavingActivity(r);
        }

        r.activity.mConfigChangeFlags |= configChanges;
        performPauseActivity(token, finished, r.isPreHoneycomb());
        if (r.isPreHoneycomb()) {
            QueuedWork.waitToFinish();
        }
        try {
            ActivityManagerNative.getDefault().activityPaused(token);
        } catch (RemoteException ex) {}
    }
}
```

接下来看函数 activityPaused，此函数在文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中定义，该函数的功能是通过 Binder 进程间通信机制进入到函数 ActivityManagerService.activityPaused 中。函数 activityPaused 的具体实现代码如下所示：

```
public final void activityPaused(IBinder token, Bundle icicle) {
    ...
    final long origId = Binder.clearCallingIdentity();
    mMainStack.activityPaused(token, icicle, false);
    ...
}
```

8.2.8　报告暂停

再看函数 activityPaused，功能是向 AmS 报告自己已经暂停完毕。

此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final void activityPaused(IBinder token, boolean timeout) {
    if (DEBUG_PAUSE)
        Slog.v(TAG, "Activity paused: token=" + token + ", timeout=" + timeout);

    ActivityRecord r = null;

    synchronized(mService) {
        int index = indexOfTokenLocked(token);
        if (index >= 0) {
            r = mHistory.get(index);
            mHandler.removeMessages(PAUSE_TIMEOUT_MSG, r);
            if (mPausingActivity == r) {
                if (DEBUG_STATES)
                    Slog.v(TAG, "Moving to PAUSED: " + r
                      + (timeout ? " (due to timeout)" : " (pause complete)"));
                r.state = ActivityState.PAUSED;
                completePauseLocked();
```

```
            } else {
                EventLog.writeEvent(EventLogTags.AM_FAILED_TO_PAUSE,
                  r.userId, System.identityHashCode(r), r.shortComponentName,
                  mPausingActivity != null?
                  mPausingActivity.shortComponentName : "(none)");
            }
        }
    }
}
```

在上述实现代码中，调用函数 indexOfTokenLocked 找到指定的 token 在 mHistory 中对应的 HistoryRecord。如果的确存在 token，那么可以直接将 token 转换为 HistoryRecord。如果函数 actvityPaused()的参数 timeout 为 false，则说明是非超时的、正常时间内暂停的 Activity，此时，需要从 mHandler 中移除还没有被处理的超时消息。

再看函数 completePauseLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void completePauseLocked() {
    ActivityRecord prev = mPausingActivity;
    if (DEBUG_PAUSE) Slog.v(TAG, "Complete pause: " + prev);

    if (prev != null) {
        if (prev.finishing) {
            if (DEBUG_PAUSE) Slog.v(TAG, "Executing finish of activity: " + prev);
            prev = finishCurrentActivityLocked(prev, FINISH_AFTER_VISIBLE, false);
        } else if (prev.app != null) {
            if (DEBUG_PAUSE) Slog.v(TAG, "Enqueueing pending stop: " + prev);
            if (prev.waitingVisible) {
                prev.waitingVisible = false;
                mWaitingVisibleActivities.remove(prev);
                if (DEBUG_SWITCH || DEBUG_PAUSE)
                    Slog.v(TAG, "Complete pause, no longer waiting: " + prev);
            }
            if (prev.configDestroy) {
                if (DEBUG_PAUSE) Slog.v(TAG, "Destroying after pause: " + prev);
                destroyActivityLocked(prev, true, false, "pause-config");
            } else {
                mStoppingActivities.add(prev);
                if (mStoppingActivities.size() > 3) {
                    if (DEBUG_PAUSE)
                        Slog.v(TAG, "To many pending stops, forcing idle");
                    scheduleIdleLocked();
                } else {
                    checkReadyForSleepLocked();
                }
            }
        } else {
            if (DEBUG_PAUSE)
```

```
                Slog.v(TAG, "App died during pause, not stopping: " + prev);
            prev = null;
        }
        mPausingActivity = null;
    }

    if (!mService.isSleeping()) {
        resumeTopActivityLocked(prev);
    } else {
        checkReadyForSleepLocked();
        ActivityRecord top = topRunningActivityLocked(null);
        if (top==null || (prev!=null && top!=prev)) {
            resumeTopActivityLocked(null);
        }
    }

    if (prev != null) {
        prev.resumeKeyDispatchingLocked();
    }

    if (prev.app!=null && prev.cpuTimeAtResume>0
      && mService.mBatteryStatsService.isOnBattery()) {
        long diff = 0;
        synchronized (mService.mProcessStatsThread) {
            diff = mService.mProcessStats.getCpuTimeForPid(prev.app.pid)
                     - prev.cpuTimeAtResume;
        }
        if (diff > 0) {
            BatteryStatsImpl bsi =
              mService.mBatteryStatsService.getActiveStatistics();
            synchronized(bsi) {
                BatteryStatsImpl.Uid.Proc ps = bsi.getProcessStatsLocked(
                  prev.info.applicationInfo.uid, prev.info.packageName);
                if (ps != null) {
                    ps.addForegroundTimeLocked(diff);
                }
            }
        }
    }

    prev.cpuTimeAtResume = 0;  //reset it
}
```

在上述代码中，首先清空变量 mPausingActivity，因为现在不需要它了，然后调用函数 resumeTopActivityLokced 继续操作。在函数 resumeTopActivityLokced 中，最终会调用函数 startSpecificActivityLocked 来实现下一步操作。

函数 startSpecificActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void startSpecificActivityLocked(ActivityRecord r,
  boolean andResume, boolean checkConfig) {
    // Is this activity's application already running?
    ProcessRecord app = mService.getProcessRecordLocked(
      r.processName, r.info.applicationInfo.uid);

    if (r.launchTime == 0) {
        r.launchTime = SystemClock.uptimeMillis();
        if (mInitialStartTime == 0) {
            mInitialStartTime = r.launchTime;
        }
    } else if (mInitialStartTime == 0) {
        mInitialStartTime = SystemClock.uptimeMillis();
    }

    if (app!=null && app.thread!=null) {
        try {
            app.addPackage(r.info.packageName);
            realStartActivityLocked(r, app, andResume, checkConfig);
            return;
        } catch (RemoteException e) {
            Slog.w(TAG, "Exception when starting activity "
              + r.intent.getComponent().flattenToShortString(), e);
        }
    }

    mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
      "activity", r.intent.getComponent(), false, false);
}
```

在上述代码中，调用函数 startProcessLocked 来检查是否存在以“process + uid”命名的进程。函数 startProcessLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
final ProcessRecord startProcessLocked(String processName,
  ApplicationInfo info, boolean knownToBeDead, int intentFlags,
  String hostingType, ComponentName hostingName, boolean allowWhileBooting,
  boolean isolated) {
    ProcessRecord app;
    if (!isolated) {
        app = getProcessRecordLocked(processName, info.uid);
    } else {
        // If this is an isolated process, it can't re-use an existing process.
        app = null;
    }
    if (DEBUG_PROCESSES)
        Slog.v(TAG, "startProcess: name=" + processName
          + " app=" + app + " knownToBeDead=" + knownToBeDead
          + " thread=" + (app != null ? app.thread : null)
```

```
        + " pid=" + (app != null ? app.pid : -1));
if (app!=null && app.pid>0) {
    if (!knownToBeDead || app.thread==null) {
        if (DEBUG_PROCESSES) Slog.v(TAG, "App already running: " + app);
        // If this is a new package in the process, add the package to the list
        app.addPackage(info.packageName);
        return app;
    } else {
        if (DEBUG_PROCESSES || DEBUG_CLEANUP) Slog.v(TAG, "App died: " + app);
        handleAppDiedLocked(app, true, true);
    }
}

String hostingNameStr = hostingName!=null?
  hostingName.flattenToShortString() : null;

if (!isolated) {
    if ((intentFlags&Intent.FLAG_FROM_BACKGROUND) != 0) {
        if (mBadProcesses.get(info.processName, info.uid) != null) {
            if (DEBUG_PROCESSES)
                Slog.v(TAG, "Bad process: " + info.uid
                  + "/" + info.processName);
            return null;
        }
    } else {
        if (DEBUG_PROCESSES)
            Slog.v(TAG, "Clearing bad process: " + info.uid
              + "/" + info.processName);
        mProcessCrashTimes.remove(info.processName, info.uid);
        if (mBadProcesses.get(info.processName, info.uid) != null) {
            EventLog.writeEvent(EventLogTags.AM_PROC_GOOD,
              UserHandle.getUserId(info.uid), info.uid, info.processName);
            mBadProcesses.remove(info.processName, info.uid);
            if (app != null) {
                app.bad = false;
            }
        }
    }
}

if (app == null) {
    app = newProcessRecordLocked(null, info, processName, isolated);
    if (app == null) {
        Slog.w(TAG, "Failed making new process record for "
          + processName + "/" + info.uid + " isolated=" + isolated);
        return null;
    }
    mProcessNames.put(processName, app.uid, app);
    if (isolated) {
```

```
            mIsolatedProcesses.put(app.uid, app);
        }
    } else {
        // If this is a new package in the process, add the package to the list
        app.addPackage(info.packageName);
    }

    if (!mProcessesReady && !isAllowedWhileBooting(info)
      && !allowWhileBooting) {
        if (!mProcessesOnHold.contains(app)) {
            mProcessesOnHold.add(app);
        }
        if (DEBUG_PROCESSES)
            Slog.v(TAG, "System not ready, putting on hold: " + app);
        return app;
    }
    startProcessLocked(app, hostingType, hostingNameStr);
    return (app.pid!=0)? app : null;
}
```

在上述代码中调用了函数 startProcessLocked，而函数 startProcessLocked 调用接口 Process.start，创建了一个新的进程，新的进程会被导入类 android.app.ActivityThread，并且执行其主函数 main，并通过函数 attach 调用了 ActivityManagerService 的远程接口 ActivityManagerProxy 中的函数 attachApplication，传入的参数是 mAppThread，这是一个 ApplicationThread 类型的 Binder 对象，其功能是实现进程之间通信。

8.2.9 建立双向连接

Android 系统中，ActivityThread.java 用 thread.attach(false)来调用 AttachApplicationLocked 函数。AttachApplicationLocked 将本进程与 Activity Manager 建立双向连接，从而使本进程得到 Activity Manager 的服务。从 attach 到 attachApplicationLocked 的调用过程为：

```
attach→attachApplication(ActivityManagerService.java 中)→attachApplicationLocked
```

下面看函数 attachApplication，功能是通过 Binder 驱动程序来到 ActivityManagerService 中的函数 attachApplication。函数 attachApplication 在文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中定义，具体实现代码如下所示：

```
public void attachApplication(IApplicationThread app) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(app.asBinder());
    mRemote.transact(ATTACH_APPLICATION_TRANSACTION, data, reply, 0);
    reply.readException();
    data.recycle();
    reply.recycle();
}
```

再看 ActivityManagerService 中的函数 attachApplication，功能是将操作转交给函数 attachApplicationLocked。此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public final void attachApplication(IApplicationThread thread) {
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final long origId = Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid);
        Binder.restoreCallingIdentity(origId);
    }
}
```

接下来看函数 attachApplicationLocked，其功能是根据 CallingPid 在 mPidsSelfLocked 找到对应的 ProcessRecord 实例 app，并将 ActvitiyThread 放置在 app.thread 中。这样应用进程和 AMS 建立起来双向连接。AM 可以使用 AIDL 接口，通过 app.thread 可以访问应用进程的对象。具体的执行过程如图 8-2 所示。

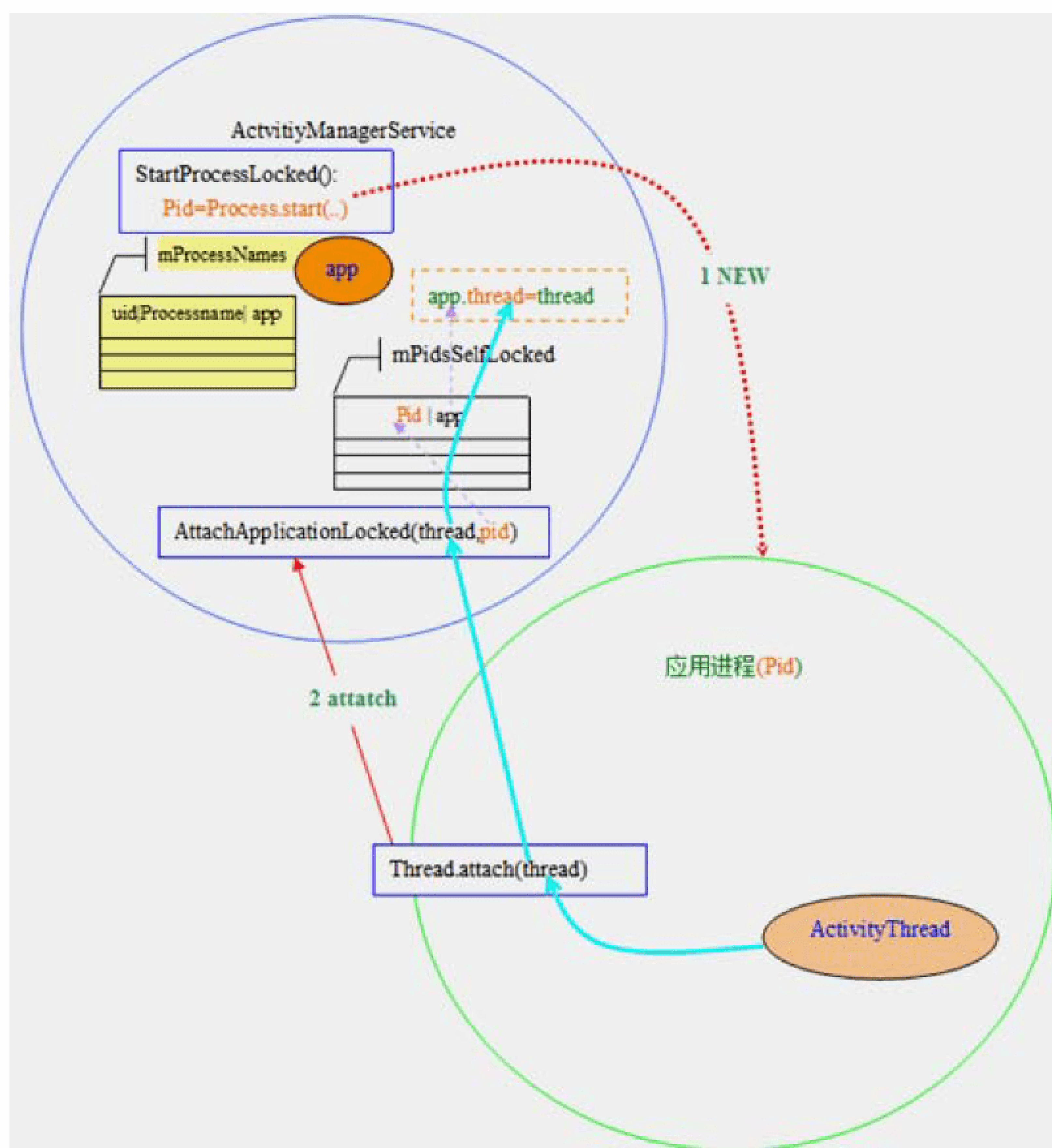


图 8-2　函数attachApplicationLocked的执行过程

函数 attachApplicationLocked 在文件 frameworks/base/services/java/com/android/server/am/

ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final boolean attachApplicationLocked(IApplicationThread thread,
  int pid) {
    ProcessRecord app;
    if (pid != MY_PID && pid >= 0) {
        synchronized (mPidsSelfLocked) {
            app = mPidsSelfLocked.get(pid);
        }
    } else {
        app = null;
    }

    if (app == null) {
        Slog.w(TAG, "No pending application record for pid " + pid
          + " (IApplicationThread " + thread + "); dropping process");
        EventLog.writeEvent(EventLogTags.AM_DROP_PROCESS, pid);
        if (pid>0 && pid!=MY_PID) {
            Process.killProcessQuiet(pid);
        } else {
            try {
                thread.scheduleExit();
            } catch (Exception e) {
                // Ignore exceptions.
            }
        }
        return false;
    }
    if (app.thread != null) {
        handleAppDiedLocked(app, true, true);
    }
    if (localLOGV)
        Slog.v(TAG, "Binding process pid " + pid + " to record " + app);

    String processName = app.processName;
    try {
        AppDeathRecipient adr = new AppDeathRecipient(app, pid, thread);
        thread.asBinder().linkToDeath(adr, 0);
        app.deathRecipient = adr;
    } catch (RemoteException e) {
        app.resetPackageList();
        startProcessLocked(app, "link fail", processName);
        return false;
    }

    EventLog.writeEvent(
      EventLogTags.AM_PROC_BOUND, app.userId, app.pid, app.processName);

    app.thread = thread;
```

```
app.curAdj = app.setAdj = -100;
app.curSchedGroup = Process.THREAD_GROUP_DEFAULT;
app.setSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
app.forcingToForeground = null;
app.foregroundServices = false;
app.hasShownUi = false;
app.debugging = false;

mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);

boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
List providers = normalMode ? generateApplicationProvidersLocked(app) : null;

if (!normalMode) {
    Slog.i(TAG, "Launching preboot mode app: " + app);
}

if (localLOGV)
    Slog.v(TAG, "New app record " + app
      + " thread=" + thread.asBinder() + " pid=" + pid);
try {
    int testMode = IApplicationThread.DEBUG_OFF;
    if (mDebugApp!=null && mDebugApp.equals(processName)) {
        testMode = mWaitForDebugger?
          IApplicationThread.DEBUG_WAIT : IApplicationThread.DEBUG_ON;
        app.debugging = true;
        if (mDebugTransient) {
            mDebugApp = mOrigDebugApp;
            mWaitForDebugger = mOrigWaitForDebugger;
        }
    }
    String profileFile = app.instrumentationProfileFile;
    ParcelFileDescriptor profileFd = null;
    boolean profileAutoStop = false;
    if (mProfileApp!=null && mProfileApp.equals(processName)) {
        mProfileProc = app;
        profileFile = mProfileFile;
        profileFd = mProfileFd;
        profileAutoStop = mAutoStopProfiler;
    }
    boolean enableOpenGlTrace = false;
    if (mOpenGlTraceApp!=null && mOpenGlTraceApp.equals(processName)) {
        enableOpenGlTrace = true;
        mOpenGlTraceApp = null;
    }
    boolean isRestrictedBackupMode = false;
    if (mBackupTarget!=null && mBackupAppName.equals(processName)) {
        isRestrictedBackupMode =
          (mBackupTarget.backupMode == BackupRecord.RESTORE)
```

```
                || (mBackupTarget.backupMode == BackupRecord.RESTORE_FULL)
                || (mBackupTarget.backupMode == BackupRecord.BACKUP_FULL);
        }

        ensurePackageDexOpt(app.instrumentationInfo != null?
          app.instrumentationInfo.packageName : app.info.packageName);
        if (app.instrumentationClass != null) {
            ensurePackageDexOpt(app.instrumentationClass.getPackageName());
        }
        if (DEBUG_CONFIGURATION)
            Slog.v(TAG, "Binding proc "
              + processName + " with config " + mConfiguration);
        ApplicationInfo appInfo = app.instrumentationInfo != null?
          app.instrumentationInfo : app.info;
        app.compat = compatibilityInfoForPackageLocked(appInfo);
        if (profileFd != null) {
            profileFd = profileFd.dup();
        }
        thread.bindApplication(
          processName, appInfo, providers,
          app.instrumentationClass, profileFile, profileFd, profileAutoStop,
          app.instrumentationArguments, app.instrumentationWatcher,
          app.instrumentationUiAutomationConnection, testMode,
          enableOpenGlTrace, isRestrictedBackupMode || !normalMode,
          app.persistent, new Configuration(mConfiguration),
          app.compat, getCommonServicesLocked(),
          mCoreSettingsObserver.getCoreSettingsLocked());
        updateLruProcessLocked(app, false);
        app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
    } catch (Exception e) {
        Slog.w(TAG, "Exception thrown during bind!", e);

        app.resetPackageList();
        app.unlinkDeathRecipient();
        startProcessLocked(app, "bind fail", processName);
        return false;
    }
    mPersistentStartingProcesses.remove(app);
    if (DEBUG_PROCESSES && mProcessesOnHold.contains(app))
        Slog.v(TAG, "Attach application locked removing on hold: " + app);
    mProcessesOnHold.remove(app);

    boolean badApp = false;
    boolean didSomething = false;
    ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
    if (hr!=null && normalMode) {
        if (hr.app == null && app.uid == hr.info.applicationInfo.uid
          && processName.equals(hr.processName)) {
            try {
```

```
                if (mHeadless) {
                    Slog.e(TAG,
                     "Starting activities not supported on headless device: " + hr);
                } else if (mMainStack.realStartActivityLocked(
                  hr, app, true, true)) {
                    didSomething = true;
                }
            } catch (Exception e) {
                Slog.w(TAG, "Exception in new application when starting activity "
                  + hr.intent.getComponent().flattenToShortString(), e);
                badApp = true;
            }
        } else {
            mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);
        }
    }
    if (!badApp) {
        try {
            didSomething |= mServices.attachApplicationLocked(app, processName);
        } catch (Exception e) {
            badApp = true;
        }
    }

    // Check if a next-broadcast receiver is in this process...
    if (!badApp && isPendingBroadcastProcessLocked(pid)) {
        try {
            didSomething = sendPendingBroadcastsLocked(app);
        } catch (Exception e) {
            badApp = true;
        }
    }
    if (!badApp && mBackupTarget!=null
      && mBackupTarget.appInfo.uid==app.uid) {
        if (DEBUG_BACKUP)
            Slog.v(TAG, "New app is backup target, launching agent for " + app);
        ensurePackageDexOpt(mBackupTarget.appInfo.packageName);
        try {
            thread.scheduleCreateBackupAgent(mBackupTarget.appInfo,
              compatibilityInfoForPackageLocked(mBackupTarget.appInfo),
              mBackupTarget.backupMode);
        } catch (Exception e) {
            Slog.w(TAG, "Exception scheduling backup agent creation: ");
            e.printStackTrace();
        }
    }

    if (badApp) {
        handleAppDiedLocked(app, false, true);
```

```
        return false;
    }
    if (!didSomething) {
        updateOomAdjLocked();
    }
    return true;
}
```

在上述代码中，先通过 pid 取回已经创建的 ProcessRecord，并放在 app 变量中；然后初始化 app 中的其他成员，最后调用函数 mMainStack.realStartActivityLocked 执行真正的 Activity 启动操作。

此处启动 Activity 的过程，是通过调用函数 mMainStack.topRunningActivityLocked(null)的方式从堆栈顶端取回来的，此时，在堆栈顶端的 Activity 就是 MainActivity。

8.2.10 启动新的Activity

函数 realStartActivityLocked 的功能是启动新的 Activity，并将新的 Activity 的相关的信息保存在参数 r 中。函数 realStartActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final boolean realStartActivityLocked(ActivityRecord r,
  ProcessRecord app, boolean andResume, boolean checkConfig)
  throws RemoteException {

    r.startFreezingScreenLocked(app, 0);
    mService.mWindowManager.setAppVisibility(r.appToken, true);
    r.startLaunchTickingLocked();
    if (checkConfig) {
        Configuration config =
          mService.mWindowManager.updateOrientationFromAppTokens(
            mService.mConfiguration,
            r.mayFreezeScreenLocked(app) ? r.appToken : null);
        mService.updateConfigurationLocked(config, r, false, false);
    }

    r.app = app;
    app.waitingToKill = null;
    r.launchCount++;
    r.lastLaunchTime = SystemClock.uptimeMillis();

    if (localLOGV) Slog.v(TAG, "Launching: " + r);

    int idx = app.activities.indexOf(r);
    if (idx < 0) {
        app.activities.add(r);
    }
    mService.updateLruProcessLocked(app, true);
```

```
    try {
        if (app.thread == null) {
            throw new RemoteException();
        }
        List<ResultInfo> results = null;
        List<Intent> newIntents = null;
        if (andResume) {
            results = r.results;
            newIntents = r.newIntents;
        }
        if (DEBUG_SWITCH) Slog.v(TAG, "Launching: " + r
          + " icicle=" + r.icicle
          + " with results=" + results + " newIntents=" + newIntents
          + " andResume=" + andResume);
        if (andResume) {
            EventLog.writeEvent(EventLogTags.AM_RESTART_ACTIVITY,
              r.userId, System.identityHashCode(r),
              r.task.taskId, r.shortComponentName);
        }
        if (r.isHomeActivity) {
            mService.mHomeProcess = app;
        }
        mService.ensurePackageDexOpt(r.intent.getComponent().getPackageName());
        r.sleeping = false;
        r.forceNewConfig = false;
        showAskCompatModeDialogLocked(r);
        r.compat =
          mService.compatibilityInfoForPackageLocked(r.info.applicationInfo);
        String profileFile = null;
        ParcelFileDescriptor profileFd = null;
        boolean profileAutoStop = false;
        if (mService.mProfileApp != null
          && mService.mProfileApp.equals(app.processName)) {
            if (mService.mProfileProc == null || mService.mProfileProc == app) {
                mService.mProfileProc = app;
                profileFile = mService.mProfileFile;
                profileFd = mService.mProfileFd;
                profileAutoStop = mService.mAutoStopProfiler;
            }
        }
        app.hasShownUi = true;
        app.pendingUiClean = true;
        if (profileFd != null) {
            try {
                profileFd = profileFd.dup();
            } catch (IOException e) {
                profileFd = null;
            }
        }
```

```
            app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
              System.identityHashCode(r), r.info,
              new Configuration(mService.mConfiguration),
              r.compat, r.icicle, results, newIntents, !andResume,
              mService.isNextTransitionForward(), profileFile, profileFd,
              profileAutoStop);

            if ((app.info.flags & ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0) {
                if (app.processName.equals(app.info.packageName)) {
                    if (mService.mHeavyWeightProcess != null
                      && mService.mHeavyWeightProcess != app) {
                        Log.w(TAG, "Starting new heavy weight process " + app
                          + " when already running " + mService.mHeavyWeightProcess);
                    }
                    mService.mHeavyWeightProcess = app;
                    Message msg = mService.mHandler.obtainMessage(
                      ActivityManagerService.POST_HEAVY_NOTIFICATION_MSG);
                    msg.obj = r;
                    mService.mHandler.sendMessage(msg);
                }
            }

        } catch (RemoteException e) {
            if (r.launchFailed) {
                Slog.e(TAG, "Second failure launching "
                  + r.intent.getComponent().flattenToShortString()
                  + ", giving up", e);
                mService.appDiedLocked(app, app.pid, app.thread);
                requestFinishActivityLocked(
                  r.appToken, Activity.RESULT_CANCELED, null, "2nd-crash", false);
                return false;
            }
            app.activities.remove(r);
            throw e;
        }

        r.launchFailed = false;
        if (updateLRUListLocked(r)) {
            Slog.w(TAG, "Activity " + r
              + " being launched, but already in LRU list");
        }

        if (andResume) {
            r.state = ActivityState.RESUMED;
            if (DEBUG_STATES)
              Slog.v(TAG, "Moving to RESUMED: " + r + " (starting new instance)");
            r.stopped = false;
            mResumedActivity = r;
            r.task.touchActiveTime();
```

```
        if (mMainStack) {
            mService.addRecentTaskLocked(r.task);
        }
        completeResumeLocked(r);
        checkReadyForSleepLocked();
        if (DEBUG_SAVED_STATE)
            Slog.i(TAG, "Launch completed; removing icicle of " + r.icicle);
    } else {
        if (DEBUG_STATES)
            Slog.v(TAG, "Moving to STOPPED: " + r
              + " (starting in stopped state)");
        r.state = ActivityState.STOPPED;
        r.stopped = true;
    }
    if (mMainStack) {
        mService.startSetupActivityLocked();
    }
    return true;
}
```

在上述代码中，参数 r 是一个 ActivityRecord 类型的 Binder 对象，用于当作这个 Activity 的 token 值。通过上述实现代码，调用 app.thread 进入到了 ApplicationThreadProxy 中的函数 scheduleLaunchActivity 中。

8.2.11　发送通知信息

接下来看函数 scheduleLaunchActivity，功能是通知应用程序它可以加载应用程序中的默认 Activity。此函数最终会把这个请求封装成一个消息，然后通过类 ActivityThread 的成员变量 mH 将这个消息加入到应用程序的消息队列中去。函数 scheduleLaunchActivity 在 frameworks/base/core/java/android/app/ApplicationThreadNative.java 文件中定义，具体实现代码如下所示：

```
public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
  ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
  List<Intent> pendingNewIntents, boolean notResumed, boolean isForward)
  throws RemoteException {
    Parcel data = Parcel.obtain();
    data.writeInterfaceToken(IApplicationThread.descriptor);
    intent.writeToParcel(data, 0);
    data.writeStrongBinder(token);
    data.writeInt(ident);
    info.writeToParcel(data, 0);
    data.writeBundle(state);
    data.writeTypedList(pendingResults);
    data.writeTypedList(pendingNewIntents);
    data.writeInt(notResumed ? 1 : 0);
    data.writeInt(isForward ? 1 : 0);
    mRemote.transact(SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION, data, null,
      IBinder.FLAG_ONEWAY);
```

```
    data.recycle();
}
```

通过上述实现代码可知，函数 scheduleLaunchActivity 最终通过 Binder 驱动程序进入到类 ApplicationThread 中的函数 scheduleLaunchActivity 中。类 ApplicationThread 中的函数 scheduleLaunchActivity 的功能是创建一个 ActivityClientRecord 实例，并且初始化它的成员变量，然后调用类 ActivityThread 的成员函数 queueOrSendMessage 实现进一步处理。

函数 scheduleLaunchActivity 在 frameworks/base/core/java/android/app/ActivityThread.java 文件中定义，具体实现代码如下所示：

```
public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
  ActivityInfo info, Configuration curConfig, CompatibilityInfo compatInfo,
  Bundle state, List<ResultInfo> pendingResults,
  List<Intent> pendingNewIntents, boolean notResumed, boolean isForward,
  String profileName, ParcelFileDescriptor profileFd,
  boolean autoStopProfiler) {
    ActivityClientRecord r = new ActivityClientRecord();
    r.token = token;
    r.ident = ident;
    r.intent = intent;
    r.activityInfo = info;
    r.compatInfo = compatInfo;
    r.state = state;
    r.pendingResults = pendingResults;
    r.pendingIntents = pendingNewIntents;
    r.startsNotResumed = notResumed;
    r.isForward = isForward;
    r.profileFile = profileName;
    r.profileFd = profileFd;
    r.autoStopProfiler = autoStopProfiler;
    updatePendingConfiguration(curConfig);
    queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
}
```

Android

第 9 章 Content Provider数据存储

在 Android 系统中，通过 Content Provider 机制可以支持在多个应用中存储和读取数据。

在本章的内容中，将详细讲解 Android 4.3 系统中 Content Provider 模块的源码，为读者步入本书后面高级知识的学习打下基础。

9.1 Content Provider基础

在 Android 系统中，Content Provider 作为应用程序四大组件之一，起到在应用程序之间共享数据的作用，同时，它还是标准的数据访问接口。在本节的内容中，将简要介绍 Content Provider 组件的基本知识。

9.1.1 Content Provider在应用程序中的架构

如果使用命令 adb shell 连接 Android 模拟器，在/data/data 目录下会看到很多以应用程序包(package)命名的文件夹，在这些文件夹中存放的是各个应用程序的数据文件。例如进入到 Android 系统日历应用程序数据目录 com.android.providers.calendar 下的 databases 文件中，会看到一个用来保存日历数据的数据库文件 calendar.db，其权限设置代码如下所示：

```
root@android:/data/data/com.android.providers.calendar/databases # ls -l
-rw-rw---- app_17   app_17     33792 2013-09-07 15:50 calendar.db
```

在上述代码“-rw-rw----”中，各个字符的具体说明如下所示。

- 最前面的符号“-”：表示这是一个普通文件。
- 接下来的三个字符“rw-”：表示此文件的所有者对这个文件可读、可写、不可执行。
- 接下来的三个字符“rw-”：表示这个文件的所有者所在的用户组的用户对这个文件可读、可写、不可执行。
- 最后的三个字符“---”：表示其他的用户对这个文件不可读写，也不可执行。
- 接下来的两个“app_17”字符串：表示这个文件的所有者和这个所有者所在的用户组的名称均为“app_17”，这是在安装应用程序时由系统分配的。在不同的系统上的这个字符串可能会不同，但是，所表示的意义是一样的。表示只有用户 ID 为 app_17 或者用户组 ID 为 app_17 的进程才可以对文件 calendar.db 进行读写操作。

Android 系统对应用程序的数据文件有严格的保护措施。当我们开发自己的应用程序时，有时会希望读取通信录里面某个联系人的手机号码或者电子邮件，以便可以对这个联系人打电话或者发送电子邮件，这时就需要读取通信录里面的联系人数据文件了。在 Android 应用程序中，很多第三方公司都推出了专业性的 API。这些 API 的目标是将开放用户数据给第三方来使用，例如 Android 系统中的通信录，它需要把联系人数据开放出来给其他应用程序使用。但是，这些数据都是各个平台自己的核心数据和核心竞争力，它们需要有保护地进行开放。为此，Android 系统特意推出了 Content Provider，它秉承了“有保护地开放自己的数据给其他应用程序使用”的理念。

Content Provider 机制在 Android 应用项目开发中的地位十分高，这是现实需求所决定的。例如，在设计大型的复杂应用的软件中，需要分模块和分层次来实现各个子功能组件，使得各个模块功能以松耦合的方式组织在一起，完成整个应用程序功能。这样做的好处如下所示：

- 便于维护和扩展应用程序的代码和功能。
- 更加适应复杂的业务环境。

如果从垂直的方向来看一个大型的应用程序软件架构，通常会分为如下所示的层次结构。

- 数据层：用来保存数据，这些数据可以用文件的方式来组织，也可以用数据库的方式来组织，甚至可以保存在网络中。
- 数据访问接口层：负责向上面的业务层提供数据，而向下管理好数据层的数据。
- 业务层：通过数据访问层来获取一些业务相关的数据，来实现自己的业务逻辑。

根据上述层次结构的描述，可以得出一个通用 Android 应用程序的架构，如图 9-1 所示。

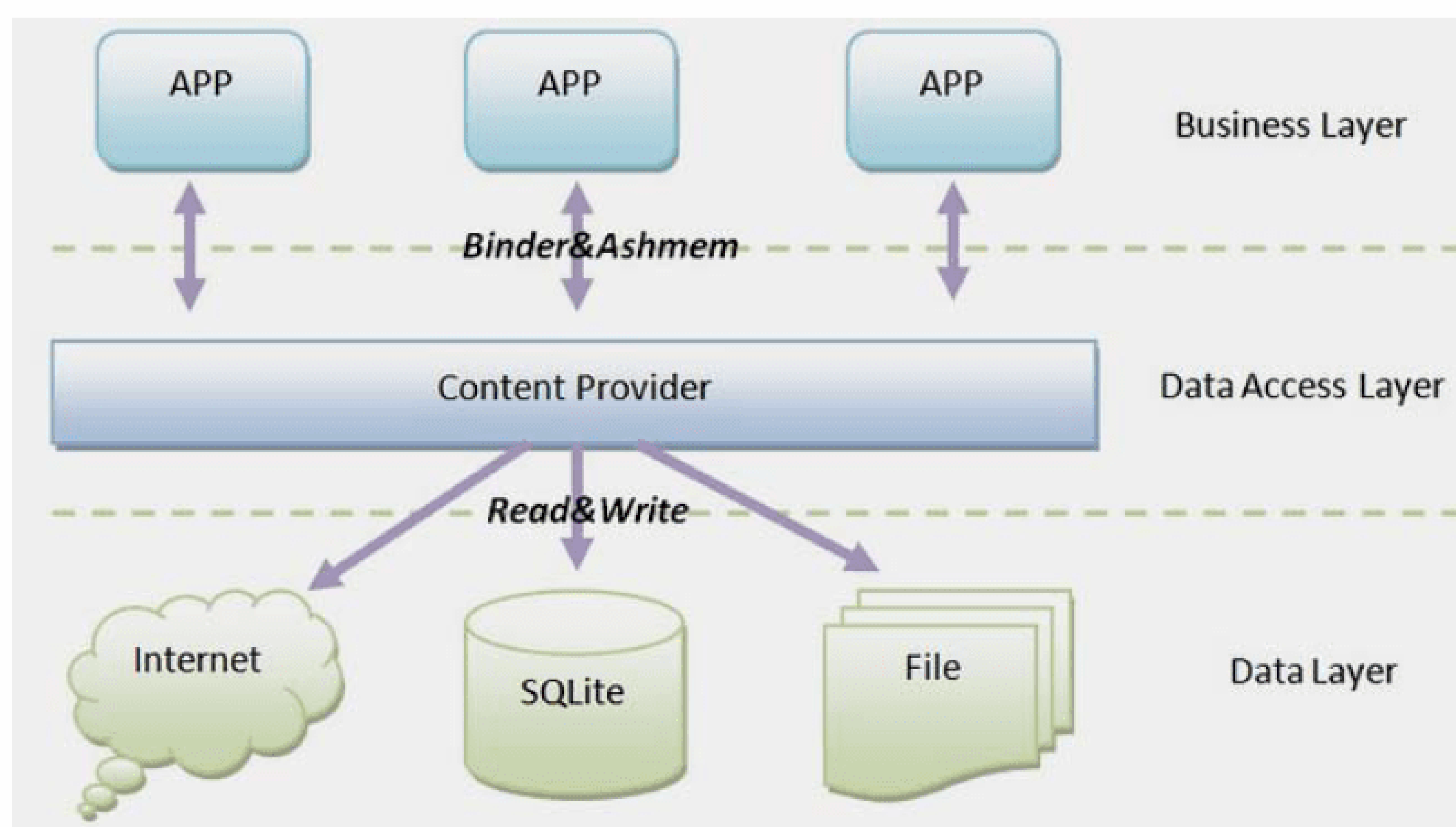


图 9-1　通用Android应用程序的架构

在图 9-1 表示的架构中，数据层视野数据库、文件或网络用来保存数据，数据访问层使用 Content Provider 来实现，而业务层就通过一些 APP 来实现。为了降低各个功能模块间的耦合度，可以把业务层的各个 APP 和数据访问层中的 Content Provider 放在不同的应用程序进程中来实现。而数据库中的数据统一由 Content Provider 来管理，Content Provider 拥有对这些文件直接进行读写的权限，并且可以根据需要有保护地把这些数据开放出来供上层 APP 使用。

9.1.2　Content Provider的常用接口

在 Android 系统中的数据是私有的，当然这些数据包括文件数据和数据库数据以及一些其他类型的数据。Android 中的两个程序之间可以进行数据交换，此功能就是通过 Content Provider 实现的。一个 Content Provider 类实现了一组标准的方法接口，从而能够让其他的应用保存或读取此 Content Provider 的各种数据类型。也就是说，一个程序可以通过实现一个 Content Provider 的抽象接口的方式将自己的数据暴露出去，而外界根本看不到，并且也不用看到这个应用暴露的数据在应用当中是如何存储的，或者是用数据库存储还是用文件存储，还是通过网上获得。这一切都不重要，重要的是外界可以通过这一套标准及统一的接口与程序里的数据打交道，可以读取程序的数据，也可以删除程序的数据。当然，中间也可能会涉及一些权限的问题。现实中比较常见的接口如下所示。

- query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)：通过 URI 进行查询，返回一个 Cursor。
- insert(Uri uri, ContentValues values)：将一组数据插入到 URI 指定的地方。

- update(Uri uri, ContentValues values, String where, String[] selectionArgs)：更新 URI 指定位置的数据。
- delete(Uri uri, String where, String[] selectionArgs)：删除指定 URI 并且符合一定条件的数据。

(1) ContentResolver 接口

外界的程序通过 ContentResolver 接口可以访问 ContentProvider 提供的数据，在 Activity 当中通过 getContentResolver()可以得到当前应用的 ContentResolver 实例。ContentResolver 提供的接口与 ContentProvider 中需要实现的接口对应，具体来说主要有以下几个。

- query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)：通过 URI 进行查询，返回一个 Cursor。
- insert(Uri uri, ContentValues values)：将一组数据插入到 URI 指定的地方。
- update(Uri uri, ContentValues values, String where, String[] selectionArgs)：更新 URI 指定位置的数据。
- delete(Uri uri, String where, String[] selectionArgs)：删除指定 URI 并且符合一定条件的数据。

(2) ContentProvider 和 ContentResolver 中的 URI

在 ContentProvider 和 ContentResolver 中，使用的 URI 的形式通常有两种，一种是指定全部数据，另一种是指定某个 ID 的数据。

我们看下面的例子：

```
content://contacts/people/      //此 URI 指定的就是全部的联系人数据
content://contacts/people/1    //此 URI 指定的是 ID 为 1 的联系人的数据
```

在上面两个类中，用到的 URI 一般由 3 部分组成，具体说明如下。

- 第一部分是：content://。
- 第二部分是：要获得数据的一个字符串片段。
- 第三部分是：ID(如果没有指定 ID，那么表示返回全部)。

因为 URI 通常比较长，而且有时候容易出错，且难以理解。所以，在 Android 中定义了一些辅助类，并且定义了一些常量来代替这些长字符串的使用，例如下面的代码：

```
Contacts.People.CONTENT_URI （联系人的 URI）
```

9.2 启动Content Provider

在 Android 系统中，Content Provider 能够为不同的应用程序访问相同的数据提供统一的入口。通常 Content Provider 运行在独立的进程中，在系统中的每个 Content Provider 只存在一个实例，其他应用程序需要在找到这个实例后才能访问它的数据。

9.2.1 获得对象接口

在启动 Content Provider 之前，首先需要一个启动参数，例如下面的代码：

```
public int getDataCount() {
    int count = 0;
    try {
        IContentProvider provider = resolver.acquireProvider(Datas.CONTENT_URI);
        Bundle bundle = provider.call(Datas.METHOD_GET_ITEM_COUNT, null, null);
        count = bundle.getInt(Datas.KEY_ITEM_COUNT, 0);
    } catch(RemoteException e) {
        e.printStackTrace();
    }
```

这样，通过调用 ContentResolver 接口 resolver 中的函数 acquireProvider，获得了与 Data.CONTENT_URI 对应的 Content Provider 对象的 IContentProvider 接口。

函数 acquireProvider 在文件 frameworks/base/core/java/android/content/ContentResolver.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireProvider(Uri uri) {
    if (!SCHEME_CONTENT.equals(uri.getScheme())) {
        return null;
    }
    final String auth = uri.getAuthority();
    if (auth != null) {
        return acquireProvider(mContext, auth);
    }
    return null;
}
```

在上述代码中，首先验证参数 URI 的 scheme 是否正确，验证此参数是否是以“content://”开头，并取出它的 authority 部分，然后调用函数 acquireProvider，完成获取 ContentProvider 接口的操作。

再看另外一个成员函数 acquireProvider，此函数在文件 frameworks/base/core/java/android/app/ContextImpl.java 中定义，具体实现代码如下所示：

```
protected IContentProvider acquireProvider(Context context, String name) {
    return mMainThread.acquireProvider(context, name);
}
```

在上述代码中，调用类 ActivityThread 中的函数 acquireProvider 来获取 Content Provider 接口。

下面看类 ActivityThread 中的函数 acquireProvider，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireProvider(Context c, String auth, int userId,
  boolean stable) {
    final IContentProvider provider =
      acquireExistingProvider(c, auth, userId, stable);
    if (provider != null) {
        return provider;
    }
```

```
    IActivityManager.ContentProviderHolder holder = null;
    try {
        holder = ActivityManagerNative.getDefault().getContentProvider(
          getApplicationThread(), auth, userId, stable);
    } catch (RemoteException ex) {}
    if (holder == null) {
        Slog.e(TAG, "Failed to find provider info for " + auth);
        return null;
    }
    holder = installProvider(c, holder, holder.info,
                             true /*noisy*/, holder.noReleaseNeeded, stable);
    return holder.provider;
}
```

在上述代码中，调用函数 getProvider 来进一步获取 Content Provider 接口。

9.2.2 存在校验

接下来看函数 acquireExistingProvider，其功能是校验本地是否已经存在所要获取的 ContentProvider 接口，如果存在，则调用函数 getProvider 直接获取。

函数 getExistingProvider 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireExistingProvider(
  Context c, String auth, int userId, boolean stable) {
    synchronized(mProviderMap) {
        final ProviderKey key = new ProviderKey(auth, userId);
        final ProviderClientRecord pr = mProviderMap.get(key);
        if (pr == null) {
            return null;
        }

        IContentProvider provider = pr.mProvider;
        IBinder jBinder = provider.asBinder();
        if (!jBinder.isBinderAlive()) {
            // The hosting process of the provider has died; we can't
            // use this one.
            Log.i(TAG, "Acquiring provider " + auth + " for user "
              + userId + ": existing object's process dead");
            handleUnstableProviderDiedLocked(jBinder, true);
            return null;
        }

        // Only increment the ref count if we have one.  If we don't then the
        // provider is not reference counted and never needs to be released.
        ProviderRefCount prc = mProviderRefCountMap.get(jBinder);
        if (prc != null) {
            incProviderRefLocked(prc, stable);
```

```
        }
        return provider;
    }
}
```

接下来看函数 getContentProvider，功能是处理类型为 GET_CONTENT_PROVIDER_TRANSACTION 的进程通信请求，并调用函数 getContentProviderImpl 实现进一步处理。函数 getContentProvider 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public final ContentProviderHolder getContentProvider(
  IApplicationThread caller, String name, int userId, boolean stable) {
    enforceNotIsolatedCaller("getContentProvider");
    if (caller == null) {
        String msg =
          "null IApplicationThread when getting content provider " + name;
        Slog.w(TAG, msg);
        throw new SecurityException(msg);
    }
    userId = handleIncomingUser(
      Binder.getCallingPid(), Binder.getCallingUid(), userId,
      false, true, "getContentProvider", null);
    return getContentProviderImpl(caller, name, null, stable, userId);
}
```

再看函数 getContentProviderImpl，其功能是获得一个 ContentProviderHolder 对象，此对象用来描述 Content Provider 的代理对象。

函数 getContentProviderImpl 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final ContentProviderHolder getContentProviderImpl(
  IApplicationThread caller, String name, IBinder token,
  boolean stable, int userId) {
    ContentProviderRecord cpr;
    ContentProviderConnection conn = null;
    ProviderInfo cpi = null;

    synchronized(this) {
        ProcessRecord r = null;
        if (caller != null) {
            //获取 ActivityManagerService 返回的与参数 name 对应的 Content Provider
            //代理对象的程序进程信息
            r = getRecordForAppLocked(caller);
            if (r == null) {
                throw new SecurityException(
                  "Unable to find app for caller " + caller
                  + " (pid=" + Binder.getCallingPid()
                  + ") when getting content provider " + name);
            }
```

```
    }

    // First check if this content provider has been published...
    cpr = mProviderMap.getProviderByName(name, userId);
    boolean providerRunning = cpr!=null;
    if (providerRunning) {
        cpi = cpr.info;
        String msg;
        if ((msg=checkContentProviderPermissionLocked(cpi, r)) != null) {
            throw new SecurityException(msg);
        }
        //查看一个这个Content Provider否配置了multiprocess属性为true,
        //是否允许在客户进程中加载
        if (r!=null && cpr.canRunHere(r)) {
            // This provider has been published or is in the process
            // of being published...  but it is also allowed to run
            // in the caller's process, so don't make a connection
            // and just let the caller instantiate its own instance.
            ContentProviderHolder holder = cpr.newHolder(null);
            holder.provider = null;
            return holder;
        }

        final long origId = Binder.clearCallingIdentity();

        conn = incProviderCountLocked(r, cpr, token, stable);
        if (conn!=null && (conn.stableCount+conn.unstableCount)==1) {
            if (cpr.proc != null
              && r.setAdj <= ProcessList.PERCEPTIBLE_APP_ADJ) {
                updateLruProcessLocked(cpr.proc, false);
            }
        }

        if (cpr.proc != null) {
            if (false) {
                if (cpr.name.flattenToShortString().equals(
                  "com.android.providers.calendar/.CalendarProvider2")) {
                    Slog.v(TAG, "****************** KILLING "
                      + cpr.name.flattenToShortString());
                    Process.killProcess(cpr.proc.pid);
                }
            }
            boolean success = updateOomAdjLocked(cpr.proc);
            if (DEBUG_PROVIDER) Slog.i(TAG, "Adjust success: " + success);
            if (!success) {
                Slog.i(TAG,
                  "Existing provider " + cpr.name.flattenToShortString()
                  + " is crashing; detaching " + r);
                boolean lastRef =
```

```
                decProviderCountLocked(conn, cpr, token, stable);
               appDiedLocked(cpr.proc, cpr.proc.pid, cpr.proc.thread);
               if (!lastRef) {
                   return null;
               }
               providerRunning = false;
               conn = null;
           }
       }

       Binder.restoreCallingIdentity(origId);
   }

   boolean singleton;
   if (!providerRunning) {
       try {
           //获得 PackageManagerService 服务接口,
           //分别获取 ArticlesProvider 应用程序的相关信息,
           //分别保存在 cpi 和 cpr 这两个本地变量中。
           cpi = AppGlobals.getPackageManager().
           resolveContentProvider(name,
             STOCK_PM_FLAGS | PackageManager.GET_URI_ PERMISSION_PATTERNS,
             userId);
       } catch (RemoteException ex) {}
       if (cpi == null) {
           return null;
       }
       singleton = isSingleton(
         cpi.processName, cpi.applicationInfo, cpi.name, cpi.flags);
       if (singleton) {
           userId = 0;
       }
       cpi.applicationInfo = getAppInfoForUser(cpi.applicationInfo, userId);

       String msg;
       if ((msg=checkContentProviderPermissionLocked(cpi, r)) != null) {
           throw new SecurityException(msg);
       }

       if (!mProcessesReady && !mDidUpdate && !mWaitingUpdate
         && !cpi.processName.equals("system")) {
           throw new IllegalArgumentException(
             "Attempt to launch content provider before system ready");
       }
       if (mStartedUsers.get(userId) == null) {
           Slog.w(TAG, "Unable to launch app "
             + cpi.applicationInfo.packageName + "/"
             + cpi.applicationInfo.uid + " for provider "
             + name + ": user " + userId + " is stopped");
```

```
            return null;
        }

        ComponentName comp = new ComponentName(cpi.packageName, cpi.name);
        cpr = mProviderMap.getProviderByClass(comp, userId);
        final boolean firstClass = cpr==null;
        if (firstClass) {
            try {
                ApplicationInfo ai = AppGlobals.getPackageManager()
                  .getApplicationInfo(cpi.applicationInfo.packageName,
                  STOCK_PM_FLAGS, userId);
                if (ai == null) {
                    Slog.w(TAG,
                      "No package info for content provider " + cpi.name);
                    return null;
                }
                ai = getAppInfoForUser(ai, userId);
                cpr = new ContentProviderRecord(this, cpi, ai, comp, singleton);
            } catch (RemoteException ex) {
                // pm is in same process, this will never happen.
            }
        }

        if (r!=null && cpr.canRunHere(r)) {
            return cpr.newHolder(null);
        }

        if (DEBUG_PROVIDER) {
            RuntimeException e = new RuntimeException("here");
            Slog.w(TAG, "LAUNCHING REMOTE PROVIDER (myuid " + r.uid
              + " pruid " + cpr.appInfo.uid + "): " + cpr.info.name, e);
        }

        //系统中所有正在加载的Content Provider都
        //保存在mLaunchingProviders成员变量中。
        //在加载相应的Content Provider之前，
        //首先要判断一下它是否正在被其他应用程序加载，
        //如果是，就不用重复加载了
        final int N = mLaunchingProviders.size();
        int i;
        for (i=0; i<N; i++) {
            if (mLaunchingProviders.get(i) == cpr) {
                break;
            }
        }

        // 条件i >= N为true，表明没有其他应用程序正在加载这个Content Provider
        if (i >= N) {
            final long origId = Binder.clearCallingIdentity();
```

```
                try {
                    // Content provider is now in use, its package can't be stopped.
                    try {
                        AppGlobals.getPackageManager().setPackageStoppedState(
                          cpr.appInfo.packageName, false, userId);
                    } catch (RemoteException e) {
                    } catch (IllegalArgumentException e) {
                        Slog.w(TAG, "Failed trying to unstop package "
                          + cpr.appInfo.packageName + ": " + e);
                    }
                    //调用 startProcessLocked 函数来启动一个新的进程
                    //来加载这个 Content Provider 对应的类,
                    //然后把这个正在加载的信息增加到 mLaunchingProviders 中去
                    ProcessRecord proc = startProcessLocked(cpi.processName,
                      cpr.appInfo, false, 0, "content provider",
                      new ComponentName(cpi.applicationInfo.packageName,
                      cpi.name), false, false);
                    if (proc == null) {
                        Slog.w(TAG, "Unable to launch app "
                          + cpi.applicationInfo.packageName + "/"
                          + cpi.applicationInfo.uid + " for provider "
                          + name + ": process is bad");
                        return null;
                    }
                    cpr.launchingApp = proc;
                    mLaunchingProviders.add(cpr);
                } finally {
                    Binder.restoreCallingIdentity(origId);
                }
            }

            if (firstClass) {
                mProviderMap.putProviderByClass(comp, cpr);
            }

            mProviderMap.putProviderByName(name, cpr);
            conn = incProviderCountLocked(r, cpr, token, stable);
            if (conn != null) {
                conn.waiting = true;
            }
        }
    }
    synchronized(cpr) {
        while (cpr.provider == null) {
            if (cpr.launchingApp == null) {
                Slog.w(TAG, "Unable to launch app "
                  + cpi.applicationInfo.packageName + "/"
                  + cpi.applicationInfo.uid + " for provider "
```

```
                + name + ": launching app became null");
            EventLog.writeEvent(EventLogTags.AM_PROVIDER_LOST_PROCESS,
              UserHandle.getUserId(cpi.applicationInfo.uid),
              cpi.applicationInfo.packageName,
              cpi.applicationInfo.uid, name);
            return null;
        }
        try {
            if (DEBUG_MU) {
                Slog.v(TAG_MU,
                  "Waiting to start provider " + cpr + " launchingApp="
                  + cpr.launchingApp);
            }
            if (conn != null) {
                conn.waiting = true;
            }
            cpr.wait();
        } catch (InterruptedException ex) {
        } finally {
            if (conn != null) {
                conn.waiting = false;
            }
        }
    }
  }
  return cpr!=null? cpr.newHolder(conn) : null;
}
```

9.2.3 启动Android应用程序

在 Android 系统中，启动 Android 应用程序的步骤是依次启动如下所示的函数：

- ActivityManagerService.startProcessLocked
- Process.start
- ActivityThread.main
- ActivityThread.attach
- ActivityManagerService.attachApplication

9.2.4 根据进程启动Content Provider

接下来看函数 attachApplicationLocked，功能是处理类型为 GET_CONTENT_PROVIDER_TRANSACTION 的进程通信请求，并启动指定 ID 的 Content Provider。

函数 attachApplicationLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final boolean attachApplicationLocked(IApplicationThread thread,
  int pid) {
```

```
ProcessRecord app;
if (pid!=MY PID && pid>=0) {
    synchronized (mPidsSelfLocked) {
        app = mPidsSelfLocked.get(pid);
    }
} else {
    app = null;
}

if (app == null) {
    Slog.w(TAG, "No pending application record for pid " + pid
      + " (IApplicationThread " + thread + "); dropping process");
    EventLog.writeEvent(EventLogTags.AM_DROP_PROCESS, pid);
    if (pid>0 && pid!=MY_PID) {
        Process.killProcessQuiet(pid);
    } else {
        try {
            thread.scheduleExit();
        } catch (Exception e) {
            // Ignore exceptions.
        }
    }
    return false;
}
if (app.thread != null) {
    handleAppDiedLocked(app, true, true);
}

// Tell the process all about itself.

if (localLOGV)
    Slog.v(TAG, "Binding process pid " + pid + " to record " + app);

String processName = app.processName;
try {
    AppDeathRecipient adr = new AppDeathRecipient(app, pid, thread);
    thread.asBinder().linkToDeath(adr, 0);
    app.deathRecipient = adr;
} catch (RemoteException e) {
    app.resetPackageList();
    startProcessLocked(app, "link fail", processName);
    return false;
}

EventLog.writeEvent(
  EventLogTags.AM_PROC_BOUND, app.userId, app.pid, app.processName);

app.thread = thread;
app.curAdj = app.setAdj = -100;
```

```
app.curSchedGroup = Process.THREAD_GROUP_DEFAULT;
app.setSchedGroup = Process.THREAD GROUP BG NONINTERACTIVE;
app.forcingToForeground = null;
app.foregroundServices = false;
app.hasShownUi = false;
app.debugging = false;

mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);

boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
List providers = normalMode? generateApplicationProvidersLocked(app) : null;

if (!normalMode) {
    Slog.i(TAG, "Launching preboot mode app: " + app);
}

if (localLOGV)
    Slog.v(TAG, "New app record " + app
      + " thread=" + thread.asBinder() + " pid=" + pid);
try {
    int testMode = IApplicationThread.DEBUG OFF;
    if (mDebugApp!=null && mDebugApp.equals(processName)) {
        testMode = mWaitForDebugger?
          IApplicationThread.DEBUG_WAIT : IApplicationThread.DEBUG_ON;
        app.debugging = true;
        if (mDebugTransient) {
            mDebugApp = mOrigDebugApp;
            mWaitForDebugger = mOrigWaitForDebugger;
        }
    }
    String profileFile = app.instrumentationProfileFile;
    ParcelFileDescriptor profileFd = null;
    boolean profileAutoStop = false;
    if (mProfileApp!=null && mProfileApp.equals(processName)) {
        mProfileProc = app;
        profileFile = mProfileFile;
        profileFd = mProfileFd;
        profileAutoStop = mAutoStopProfiler;
    }
    boolean enableOpenGlTrace = false;
    if (mOpenGlTraceApp!== null && mOpenGlTraceApp.equals(processName)) {
        enableOpenGlTrace = true;
        mOpenGlTraceApp = null;
    }

    // If the app is being launched for restore or full backup, set it up specially
    boolean isRestrictedBackupMode = false;
    if (mBackupTarget!=null && mBackupAppName.equals(processName)) {
        isRestrictedBackupMode =
```

```
            (mBackupTarget.backupMode == BackupRecord.RESTORE)
            || (mBackupTarget.backupMode == BackupRecord.RESTORE_FULL)
            || (mBackupTarget.backupMode == BackupRecord.BACKUP_FULL);
    }

    ensurePackageDexOpt(app.instrumentationInfo!=null?
      app.instrumentationInfo.packageName : app.info.packageName);
    if (app.instrumentationClass != null) {
        ensurePackageDexOpt(app.instrumentationClass.getPackageName());
    }
    if (DEBUG_CONFIGURATION)
      Slog.v(TAG, "Binding proc " + processName + " with config " + mConfiguration);
    ApplicationInfo appInfo = app.instrumentationInfo != null?
      app.instrumentationInfo : app.info;
    app.compat = compatibilityInfoForPackageLocked(appInfo);
    if (profileFd != null) {
        profileFd = profileFd.dup();
    }
    thread.bindApplication(processName, appInfo, providers,
      app.instrumentationClass, profileFile, profileFd, profileAutoStop,
      app.instrumentationArguments, app.instrumentationWatcher,
      app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,
      isRestrictedBackupMode || !normalMode, app.persistent,
      new Configuration(mConfiguration), app.compat,
      getCommonServicesLocked(),
      mCoreSettingsObserver.getCoreSettingsLocked());
    updateLruProcessLocked(app, false);
    app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
} catch (Exception e) {
    Slog.w(TAG, "Exception thrown during bind!", e);
    app.resetPackageList();
    app.unlinkDeathRecipient();
    startProcessLocked(app, "bind fail", processName);
    return false;
}
mPersistentStartingProcesses.remove(app);
if (DEBUG_PROCESSES && mProcessesOnHold.contains(app))
    Slog.v(TAG, "Attach application locked removing on hold: " + app);
mProcessesOnHold.remove(app);

boolean badApp = false;
boolean didSomething = false;
ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
if (hr!=null && normalMode) {
    if (hr.app == null && app.uid == hr.info.applicationInfo.uid
      && processName.equals(hr.processName)) {
        try {
            if (mHeadless) {
                Slog.e(TAG,
```

```
                        "Starting activities not supported on headless device: " + hr);
                } else if (mMainStack.realStartActivityLocked(hr,
                  app, true, true)) {
                    didSomething = true;
                }
            } catch (Exception e) {
                Slog.w(TAG, "Exception in new application when starting activity "
                  + hr.intent.getComponent().flattenToShortString(), e);
                badApp = true;
            }
        } else {
            mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);
        }
    }
    if (!badApp) {
        try {
            didSomething |= mServices.attachApplicationLocked(app, processName);
        } catch (Exception e) {
            badApp = true;
        }
    }
    if (!badApp && isPendingBroadcastProcessLocked(pid)) {
        try {
            didSomething = sendPendingBroadcastsLocked(app);
        } catch (Exception e) {
            badApp = true;
        }
    }
    if (!badApp && mBackupTarget != null
      && mBackupTarget.appInfo.uid == app.uid) {
        if (DEBUG_BACKUP)
          Slog.v(TAG, "New app is backup target, launching agent for " + app);
        ensurePackageDexOpt(mBackupTarget.appInfo.packageName);
        try {
            thread.scheduleCreateBackupAgent(mBackupTarget.appInfo,
              compatibilityInfoForPackageLocked(mBackupTarget.appInfo),
              mBackupTarget.backupMode);
        } catch (Exception e) {
            Slog.w(TAG, "Exception scheduling backup agent creation: ");
            e.printStackTrace();
        }
    }
    if (badApp) {
        handleAppDiedLocked(app, false, true);
        return false;
    }

    if (!didSomething) {
        updateOomAdjLocked();
```

```
    }
    return true;
}
```

在上述代码中，调用函数 generateApplicationProvidersLocked 获取了需要在 ProcessRecord 对象 app 描述的程序进程中启动的 Content Provider 组件。函数 generateApplicationProvidersLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final List<ProviderInfo>
  generateApplicationProvidersLocked(ProcessRecord app) {
    List<ProviderInfo> providers = null;
    try {
        providers = AppGlobals.getPackageManager()
          .queryContentProviders(app.processName, app.uid,
          STOCK_PM_FLAGS | PackageManager.GET_URI_PERMISSION_PATTERNS);
    } catch (RemoteException ex) {}
    if (DEBUG_MU)
        Slog.v(TAG_MU,
          "generateApplicationProvidersLocked, app.info.uid = " + app.uid);
    int userId = app.userId;
    if (providers != null) {
        int N = providers.size();
        for (int i=0; i<N; i++) {
            ProviderInfo cpi = (ProviderInfo)providers.get(i);
            boolean singleton = isSingleton(cpi.processName, cpi.applicationInfo,
              cpi.name, cpi.flags);
            if (singleton && UserHandle.getUserId(app.uid) != 0) {
                providers.remove(i);
                N--;
                continue;
            }

            ComponentName comp = new ComponentName(cpi.packageName, cpi.name);
            ContentProviderRecord cpr =
              mProviderMap.getProviderByClass(comp, userId);
            if (cpr == null) {
                cpr = new ContentProviderRecord(
                  this, cpi, app.info, comp, singleton);
                mProviderMap.putProviderByClass(comp, cpr);
            }
            if (DEBUG_MU)
                Slog.v(TAG_MU,
                  "generateApplicationProvidersLocked, cpi.uid = " + cpr.uid);
            app.pubProviders.put(cpi.name, cpr);
            app.addPackage(cpi.applicationInfo.packageName);
            ensurePackageDexOpt(cpi.applicationInfo.packageName);
        }
    }
```

```
    return providers;
}
```

9.2.5 处理消息

再看函数 bindApplication，功能是把相关的信息都封装成一个 AppBindData 对象，然后以一个消息的形式发送到主线程的消息队列中去等待处理。这个消息最终在类 ActivityThread 的函数 handleBindApplication 中进行处理。函数 bindApplication 在文件 frameworks\base\core\java\android\app\ApplicationThreadNative.java 中定义，具体实现代码如下所示：

```
public final void bindApplication(String packageName, ApplicationInfo info,
  List<ProviderInfo> providers, ComponentName testName, String profileName,
  ParcelFileDescriptor profileFd, boolean autoStopProfiler, Bundle testArgs,
  IInstrumentationWatcher testWatcher,
  IUiAutomationConnection uiAutomationConnection, int debugMode,
  boolean openGlTrace, boolean restrictedBackupMode, boolean persistent,
  Configuration config, CompatibilityInfo compatInfo,
  Map<String, IBinder> services, Bundle coreSettings) throws RemoteException {
    Parcel data = Parcel.obtain();
    data.writeInterfaceToken(IApplicationThread.descriptor);
    data.writeString(packageName);
    info.writeToParcel(data, 0);
    data.writeTypedList(providers);
    if (testName == null) {
        data.writeInt(0);
    } else {
        data.writeInt(1);
        testName.writeToParcel(data, 0);
    }
    data.writeString(profileName);
    if (profileFd != null) {
        data.writeInt(1);
        profileFd.writeToParcel(data, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
    } else {
        data.writeInt(0);
    }
    data.writeInt(autoStopProfiler? 1 : 0);
    data.writeBundle(testArgs);
    data.writeStrongInterface(testWatcher);
    data.writeStrongInterface(uiAutomationConnection);
    data.writeInt(debugMode);
    data.writeInt(openGlTrace? 1 : 0);
    data.writeInt(restrictedBackupMode? 1 : 0);
    data.writeInt(persistent? 1 : 0);
    config.writeToParcel(data, 0);
    compatInfo.writeToParcel(data, 0);
    data.writeMap(services);
    data.writeBundle(coreSettings);
```

```
    mRemote.transact(
      BIND_APPLICATION_TRANSACTION, data, null, IBinder.FLAG_ONEWAY);
    data.recycle();
}
```

我们再看文件 frameworks\base\core\java\android\app\ApplicationThreadNative.java 中的函数 bindApplication，具体实现代码如下所示：

```
public final void bindApplication(String processName,
  ApplicationInfo appInfo, List<ProviderInfo> providers,
  ComponentName instrumentationName, String profileFile,
  ParcelFileDescriptor profileFd, boolean autoStopProfiler,
  Bundle instrumentationArgs, IInstrumentationWatcher instrumentationWatcher,
  IUiAutomationConnection instrumentationUiConnection, int debugMode,
  boolean enableOpenGlTrace, boolean isRestrictedBackupMode, boolean persistent,
  Configuration config, CompatibilityInfo compatInfo,
  Map<String, IBinder> services, Bundle coreSettings) {

    if (services != null) {
        ServiceManager.initServiceCache(services);
    }

    setCoreSettings(coreSettings);

    AppBindData data = new AppBindData();
    data.processName = processName;
    data.appInfo = appInfo;
    data.providers = providers;
    data.instrumentationName = instrumentationName;
    data.instrumentationArgs = instrumentationArgs;
    data.instrumentationWatcher = instrumentationWatcher;
    data.instrumentationUiAutomationConnection = instrumentationUiConnection;
    data.debugMode = debugMode;
    data.enableOpenGlTrace = enableOpenGlTrace;
    data.restrictedBackupMode = isRestrictedBackupMode;
    data.persistent = persistent;
    data.config = config;
    data.compatInfo = compatInfo;
    data.initProfileFile = profileFile;
    data.initProfileFd = profileFd;
    data.initAutoStopProfiler = false;
    queueOrSendMessage(H.BIND_APPLICATION, data);
}
```

9.2.6　具体启动

接下来，开始步入具体的启动步骤。首先看函数 handleBindApplication，功能是将消息中的 Content Provider 组件启动起来。函数 handleBindApplication 在文件 frameworks/base/core/java/

android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private final void handleBindApplication(AppBindData data) {
    ...
    List<ProviderInfo> providers = data.providers;
    if (providers != null) {
        installContentProviders(app, providers);
        ...
    }
```

再看函数 installContentProviders，功能是启动这些 Content Provider 组件，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private void installContentProviders(
  Context context, List<ProviderInfo> providers) {
    final ArrayList<IActivityManager.ContentProviderHolder> results =
      new ArrayList<IActivityManager.ContentProviderHolder>();

    for (ProviderInfo cpi : providers) {
        if (DEBUG_PROVIDER) {
            StringBuilder buf = new StringBuilder(128);
            buf.append("Pub ");
            buf.append(cpi.authority);
            buf.append(": ");
            buf.append(cpi.name);
            Log.i(TAG, buf.toString());
        }
        IActivityManager.ContentProviderHolder cph =
          installProvider(context, null, cpi,
          false /*noisy*/, true /*noReleaseNeeded*/, true /*stable*/);
        if (cph != null) {
            cph.noReleaseNeeded = true;
            results.add(cph);
        }
    }

    try {
        ActivityManagerNative.getDefault().publishContentProviders(
          getApplicationThread(), results);
    } catch (RemoteException ex) {}
}
```

由此可见，函数 installContentProviders 调用 installProvider 在本地安装了每一个 Content Provider 的信息，并且为每一个 Content Provider 创建了一个 ContentProviderHolder 对象来保存相关的信息。ContentProviderHolder 对象是一个 Binder 对象，功能是把 Content Provider 的信息传递给 ActivityManagerService 服务。

当处理完 Content Provider 后，还需要调用 ActivityManagerService 服务中的函数 publishContentProviders 来通知 ActivityManagerService 服务在这个进程中需要加载的 Content

Provider。

下面看函数 attachInfo，功能是初始化前面创建的 Content Provider 组件。此函数在文件 frameworks/base/core/java/android/content/ContentProvider.java 中定义，具体实现代码如下所示：

```
private void attachInfo(Context context, ProviderInfo info, boolean testing) {
    /*
    * We may be using AsyncTask from binder threads.  Make it init here
    * so its static handler is on the main thread.
    */
    AsyncTask.init();

    mNoPerms = testing;

    if (mContext == null) {
        mContext = context;
        mMyUid = Process.myUid();
        if (info != null) {
            setReadPermission(info.readPermission);
            setWritePermission(info.writePermission);
            setPathPermissions(info.pathPermissions);
            mExported = info.exported;
        }
        ContentProvider.this.onCreate();
    }
}
```

在上述代码中，根据 Content Provider 的信息 info 来设置相应的读写权限，然后调用其子类中的函数 onCreate 让子类执行初始化工作。

接下来，需要我们自定义编写函数 onCreate 实现业务初始化操作，例如下面的代码：

```
public boolean onCreate() {
    Context context = getContext();
    resolver = context.getContentResolver();
    dbHelper = new DBHelper(context, DB_NAME, null, DB_VERSION);
    return true;
}
```

最后看函数 publishContentProviders，功能是将刚刚启动的 Content Provider 的接口发布到 ActivityManagerService 服务中。函数 publishContentProviders 在文件 frameworks/base/core/java/android/app/ActivityManageNative.java 中定义，具体实现代码如下所示：

```
public void publishContentProviders(IApplicationThread caller,
  List<ContentProviderHolder> providers) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller!=null? caller.asBinder() : null);
    data.writeTypedList(providers);
    mRemote.transact(PUBLISH_CONTENT_PROVIDERS_TRANSACTION, data, reply, 0);
```

```
    reply.readException();
    data.recycle();
    reply.recycle();
}
```

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 publishContentProviders，具体实现代码如下所示：

```
public final void publishContentProviders(IApplicationThread caller,
  List<ContentProviderHolder> providers) {
    if (providers == null) {
        return;
    }

    enforceNotIsolatedCaller("publishContentProviders");
    synchronized (this) {
        final ProcessRecord r = getRecordForAppLocked(caller);
        if (DEBUG_MU)
            Slog.v(TAG_MU, "ProcessRecord uid = " + r.uid);
        if (r == null) {
            throw new SecurityException(
              "Unable to find app for caller " + caller
              + " (pid=" + Binder.getCallingPid()
              + ") when publishing content providers");
        }

        final long origId = Binder.clearCallingIdentity();

        final int N = providers.size();
        for (int i=0; i<N; i++) {
            ContentProviderHolder src = providers.get(i);
            if (src == null || src.info == null || src.provider == null) {
                continue;
            }
            ContentProviderRecord dst = r.pubProviders.get(src.info.name);
            if (DEBUG_MU)
                Slog.v(TAG_MU, "ContentProviderRecord uid = " + dst.uid);
            if (dst != null) {
                ComponentName comp =
                  new ComponentName(dst.info.packageName, dst.info.name);
                //把这个 Content Provider 信息
                //保存在 mProvidersByClass 和 mProvidersByName 中
                mProviderMap.putProviderByClass(comp, dst);
                String names[] = dst.info.authority.split(";");
                for (int j=0; j<names.length; j++) {
                    mProviderMap.putProviderByName(names[j], dst);
                }
                //因为这个 Content Provider 已经加载好了,
                //因此，把它从 mLaunchingProviders 列表中删除
```

```
                int NL = mLaunchingProviders.size();
                int j;
                for (j=0; j<NL; j++) {
                    if (mLaunchingProviders.get(j) == dst) {
                        mLaunchingProviders.remove(j);
                        j--;
                        NL--;
                    }
                }
                //设置这个 ContentProviderRecord 对象 dst 的 provider 域
                //为从参数传进来的 Content Provider 远程接口
                synchronized (dst) {
                    dst.provider = src.provider;
                    dst.proc = r;
                    dst.notifyAll();
                }
                updateOomAdjLocked(r);
            }
        }

        Binder.restoreCallingIdentity(origId);
    }
}
```

9.3　Content Provider数据共享

在 Android 系统中，Content Provider 的最核心功能便是数据共享，提供了一种保存数据的作用。Content Provider 组件可以在不同的应用程序之间传输数据，这一功能是基于匿名共享内存机制来实现的。在 Android 应用程序中，通常将共享数据保存在 SQLite 中，Content Provider 借助于 SQLite 数据库游标(SQLiteCursor)来实现最终的数据共享。在本节的内容中，将详细分析 Content Provider 实现数据共享的源码。

9.3.1　获取接口

首先看函数 query，功能是调用函数 acquireProvider 获得与参数 uri 对应的 Content Provider 接口，并通过这个接口中的函数 query 来获取相应的数据。函数 query 在文件 frameworks/base/core/java/android/content/ContentResolver.java 中定义，具体实现代码如下所示：

```
public final Cursor query(final Uri uri, String[] projection,
  String selection, String[] selectionArgs, String sortOrder,
  CancellationSignal cancellationSignal) {
    IContentProvider unstableProvider = acquireUnstableProvider(uri);
    if (unstableProvider == null) {
        return null;
    }
```

```
    IContentProvider stableProvider = null;
    Cursor qCursor = null;
    try {
        long startTime = SystemClock.uptimeMillis();

        ICancellationSignal remoteCancellationSignal = null;
        if (cancellationSignal != null) {
            cancellationSignal.throwIfCanceled();
            remoteCancellationSignal =
              unstableProvider.createCancellationSignal();
            cancellationSignal.setRemote(remoteCancellationSignal);
        }
        try {
            qCursor = unstableProvider.query(mPackageName, uri, projection,
              selection, selectionArgs, sortOrder, remoteCancellationSignal);
        } catch (DeadObjectException e) {
            // The remote process has died...  but we only hold an unstable
            // reference though, so we might recover!!!  Let's try!!!!
            // This is exciting!!1!!1!!!!1
            unstableProviderDied(unstableProvider);
            //获得与参数uri对应的Content Provider接口
            stableProvider = acquireProvider(uri);
            if (stableProvider == null) {
                return null;
            }
            qCursor = stableProvider.query(mPackageName, uri, projection,
              selection, selectionArgs, sortOrder, remoteCancellationSignal);
        }
        if (qCursor == null) {
            return null;
        }

        // Force query execution. Might fail and throw a runtime exception here.
        qCursor.getCount();
        long durationMillis = SystemClock.uptimeMillis() - startTime;
        maybeLogQueryToEventLog(
          durationMillis, uri, projection, selection, sortOrder);

        // Wrap the cursor object into CursorWrapperInner object.
        CursorWrapperInner wrapper = new CursorWrapperInner(qCursor,
          stableProvider!=null? stableProvider : acquireProvider(uri));
        stableProvider = null;
        qCursor = null;
        return wrapper;
    } catch (RemoteException e) {
        // Arbitrary and not worth documenting, as Activity
        // Manager will kill this process shortly anyway.
        return null;
    } finally {
```

```
        if (qCursor != null) {
            qCursor.close();
        }
        if (unstableProvider != null) {
            releaseUnstableProvider(unstableProvider);
        }
        if (stableProvider != null) {
            releaseProvider(stableProvider);
        }
    }
}
```

在上述代码中，会调用返回来的 Content Provider 接口来获取数据。这个 Content Provider 接口实际上是在类 ContentProvider 内部所创建的一个 Transport 对象的远程接口。类 Transport 继承于类 ContentProviderNative，是一个 Binder 对象的 Stub 类。

接下来需要依次调用如下所示的函数：

- ContentResolver.acquireProvider
- ApplicationContentResolver.acquireProvider
- ActivityThread.acquireProvider
- ActivityThread.getProvider

接下来，进入到这个 Binder 对象的 Proxy 类 ContentProviderProxy 中执行函数 query。在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中，函数 query 的具体实现代码如下所示：

```
public Cursor query(String callingPkg, Uri url, String[] projection,
  String selection, String[] selectionArgs, String sortOrder,
  ICancellationSignal cancellationSignal) throws RemoteException {
    BulkCursorToCursorAdaptor adaptor = new BulkCursorToCursorAdaptor();
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    try {
        data.writeInterfaceToken(IContentProvider.descriptor);

        data.writeString(callingPkg);
        url.writeToParcel(data, 0);
        int length = 0;
        if (projection != null) {
            length = projection.length;
        }
        data.writeInt(length);
        for (int i=0; i<length; i++) {
            data.writeString(projection[i]);
        }
        data.writeString(selection);
        if (selectionArgs != null) {
            length = selectionArgs.length;
        } else {
```

```
            length = 0;
        }
        data.writeInt(length);
        for (int i=0; i<length; i++) {
            data.writeString(selectionArgs[i]);
        }
        data.writeString(sortOrder);
        data.writeStrongBinder(adaptor.getObserver().asBinder());
        data.writeStrongBinder(cancellationSignal!=null?
          cancellationSignal.asBinder() : null);

        mRemote.transact(IContentProvider.QUERY_TRANSACTION, data, reply, 0);

        DatabaseUtils.readExceptionFromParcel(reply);

        if (reply.readInt() != 0) {
            BulkCursorDescriptor d =
              BulkCursorDescriptor.CREATOR.createFromParcel(reply);
            adaptor.initialize(d);
        } else {
            adaptor.close();
            adaptor = null;
        }
        return adaptor;
    } catch (RemoteException ex) {
        adaptor.close();
        throw ex;
    } catch (RuntimeException ex) {
        adaptor.close();
        throw ex;
    } finally {
        data.recycle();
        reply.recycle();
    }
}
```

在上述代码中，先创建了一个 CursorWindow 对象，在此 CursorWindow 对象中包含了一块匿名共享内存，其作用是把这块匿名共享内存通过 Binder 进程间通信机制传给 Content Provider，这样就可以让 Content Provider 在里面返回所请求的数据。

9.3.2 创建CursorWindow对象

我们首先来看在文件 frameworks/base/core/java/android/database/CursorWindow.java 中对类 CursorWindow 的定义，主要代码如下所示：

```
public class CursorWindow extends SQLiteClosable implements Parcelable {
    private static final String STATS_TAG = "CursorWindowStats";
    private static final int sCursorWindowSize =
```

```
    Resources.getSystem().getInteger(
      com.android.internal.R.integer.config cursorWindowSize) * 1024;
  public int mWindowPtr;

  private int mStartPos;
  private final String mName;

  private final CloseGuard mCloseGuard = CloseGuard.get();

  private static native int nativeCreate(String name, int cursorWindowSize);
  private static native int nativeCreateFromParcel(Parcel parcel);
  private static native void nativeDispose(int windowPtr);
  private static native void nativeWriteToParcel(int windowPtr, Parcel parcel);

  private static native void nativeClear(int windowPtr);

  private static native int nativeGetNumRows(int windowPtr);
  private static native boolean nativeSetNumColumns(int windowPtr, int columnNum);
  private static native boolean nativeAllocRow(int windowPtr);
  private static native void nativeFreeLastRow(int windowPtr);

  private static native int nativeGetType(int windowPtr, int row, int column);
  private static native byte[] nativeGetBlob(int windowPtr, int row, int column);
  private static native String nativeGetString(int windowPtr, int row, int column);
  private static native long nativeGetLong(int windowPtr, int row, int column);
  private static native double nativeGetDouble(int windowPtr, int row, int column);
  private static native void nativeCopyStringToBuffer(
    int windowPtr, int row, int column, CharArrayBuffer buffer);

  private static native boolean nativePutBlob(
    int windowPtr, byte[] value, int row, int column);
  private static native boolean nativePutString(
    int windowPtr, String value, int row, int column);
  private static native boolean nativePutLong(
    int windowPtr, long value, int row, int column);
  private static native boolean nativePutDouble(
    int windowPtr, double value, int row, int column);
  private static native boolean nativePutNull(int windowPtr, int row, int column);

  private static native String nativeGetName(int windowPtr);
  public CursorWindow(String name)
```

在上述代码中，调用了本地函数 native_init 来实现初始化的工作，初始化的过程就是创建匿名共享内存的过程。在过程中传进来的参数 localWindow 的值为 false，表示这个匿名共享内存只能通过远程调用来访问，可以通过 Content Provider 返回的接口 Cursor 来访问这块匿名共享内存里面的数据。

本地函数 native_init 与文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中的函数 JNINativeMethod 相对应，具体实现代码如下所示：

```
static JNINativeMethod sMethods[] =
{
    /* name, signature, funcPtr */
    { "nativeCreate", "(Ljava/lang/String;I)I",
            (void*)nativeCreate },
    { "nativeCreateFromParcel", "(Landroid/os/Parcel;)I",
            (void*)nativeCreateFromParcel },
    { "nativeDispose", "(I)V",
            (void*)nativeDispose },
    { "nativeWriteToParcel", "(ILandroid/os/Parcel;)V",
            (void*)nativeWriteToParcel },
    { "nativeGetName", "(I)Ljava/lang/String;",
            (void*)nativeGetName },
    { "nativeClear", "(I)V",
            (void*)nativeClear },
    { "nativeGetNumRows", "(I)I",
            (void*)nativeGetNumRows },
    { "nativeSetNumColumns", "(II)Z",
            (void*)nativeSetNumColumns },
    { "nativeAllocRow", "(I)Z",
            (void*)nativeAllocRow },
    { "nativeFreeLastRow", "(I)V",
            (void*)nativeFreeLastRow },
    { "nativeGetType", "(III)I",
            (void*)nativeGetType },
    { "nativeGetBlob", "(III)[B",
            (void*)nativeGetBlob },
    { "nativeGetString", "(III)Ljava/lang/String;",
            (void*)nativeGetString },
    { "nativeGetLong", "(III)J",
            (void*)nativeGetLong },
    { "nativeGetDouble", "(III)D",
            (void*)nativeGetDouble },
    { "nativeCopyStringToBuffer", "(IIILandroid/database/CharArrayBuffer;)V",
            (void*)nativeCopyStringToBuffer },
    { "nativePutBlob", "(I[BII)Z",
            (void*)nativePutBlob },
    { "nativePutString", "(ILjava/lang/String;II)Z",
            (void*)nativePutString },
    { "nativePutLong", "(IJII)Z",
            (void*)nativePutLong },
    { "nativePutDouble", "(IDII)Z",
            (void*)nativePutDouble },
    { "nativePutNull", "(III)Z",
            (void*)nativePutNull },
};
```

在上述代码中调用了函数 nativeCreate，此函数在 C++层创建了一个 CursorWindow 对象，

并通过调用宏 LOG_WINDOW 将这个 C++层的 CursorWindow 对象与 Java 层的 CursorWindow 对象关联起来。

函数 nativeCreate 也是在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static jint nativeCreate(JNIEnv *env, jclass clazz, jstring nameObj,
  jint cursorWindowSize) {
    String8 name;
    const char *nameStr = env->GetStringUTFChars(nameObj, NULL);
    name.setTo(nameStr);
    env->ReleaseStringUTFChars(nameObj, nameStr);

    CursorWindow *window;
    status_t status = CursorWindow::create(name, cursorWindowSize, &window);
    if (status || !window) {
        ALOGE("Could not allocate CursorWindow '%s' of size %d due to error %d.",
          name.string(), cursorWindowSize, status);
        return 0;
    }

    LOG_WINDOW("nativeInitializeEmpty: window = %p", window);
    return reinterpret_cast<jint>(window);
}
```

再看函数 register_android_database_CursorWindow，功能是初始化用于描述 Java 层的类 CursorWindowc 的成员变量 nWindow 在类内部的偏移量。函数 register_android_database_CursorWindow 也是在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
int register_android_database_CursorWindow(JNIEnv *env)
{
    jclass clazz;
    FIND_CLASS(clazz, "android/database/CharArrayBuffer");

    GET_FIELD_ID(gCharArrayBufferClassInfo.data, clazz, "data", "[C");
    GET_FIELD_ID(gCharArrayBufferClassInfo.sizeCopied, clazz, "sizeCopied", "I");

    gEmptyString = jstring(env->NewGlobalRef(env->NewStringUTF("")));
    LOG_FATAL_IF(!gEmptyString, "Unable to create empty string");

    return AndroidRuntime::registerNativeMethods(env,
      "android/database/CursorWindow", sMethods, NELEM(sMethods));
}
```

9.3.3　数据传递

在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中的函数 query

中，通过如下代码将创建的匿名共享内存传递给了我们的应用组件，这样，应用组件可以与参数 url 对应的信息保存在里面：

```
BulkCursorDescriptor d = BulkCursorDescriptor.CREATOR.createFromParcel(reply);
```

类 BulkCursorDescriptor 在文件 frameworks\base\core\java\android\database\BulkCursorDescriptor.java 中定义，具体实现代码如下所示：

```
public final class BulkCursorDescriptor implements Parcelable {
    public static final Parcelable.Creator<BulkCursorDescriptor> CREATOR =
      new Parcelable.Creator<BulkCursorDescriptor>() {
        @Override
        public BulkCursorDescriptor createFromParcel(Parcel in) {
            BulkCursorDescriptor d = new BulkCursorDescriptor();
            d.readFromParcel(in);
            return d;
        }

        @Override
        public BulkCursorDescriptor[] newArray(int size) {
            return new BulkCursorDescriptor[size];
        }
    };

    public IBulkCursor cursor;
    public String[] columnNames;
    public boolean wantsAllOnMoveCalls;
    public int count;
    public CursorWindow window;

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeStrongBinder(cursor.asBinder());
        out.writeStringArray(columnNames);
        out.writeInt(wantsAllOnMoveCalls ? 1 : 0);
        out.writeInt(count);
        if (window != null) {
            out.writeInt(1);
            window.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }
    }
```

```
    public void readFromParcel(Parcel in) {
        cursor = BulkCursorNative.asInterface(in.readStrongBinder());
        columnNames = in.readStringArray();
        wantsAllOnMoveCalls = in.readInt() != 0;
        count = in.readInt();
        if (in.readInt() != 0) {
            window = CursorWindow.CREATOR.createFromParcel(in);
        }
    }
}
```

在上述代码中，调用 window 中的函数 writeToParcel 把 window 对象内部的匿名共享内存块，通过 Binder 进程间通信机制传输给 Content Provider 使用。当传进来的参数 adaptor 不为 null 时，会向 data 中写入整数 1，表示让 Content Provider 返回查询得到数据的元信息。

在文件 frameworks/base/core/java/android/database/CursorWindow.java 中，函数 writeToParcel 的具体实现代码如下所示：

```
public void writeToParcel(Parcel dest, int flags) {
    acquireReference();
    try {
        dest.writeInt(mStartPos);
        nativeWriteToParcel(mWindowPtr, dest);
    } finally {
        releaseReference();
    }
    if ((flags & Parcelable.PARCELABLE_WRITE_RETURN_VALUE) != 0) {
        releaseReference();
    }
}
```

在上述代码中，调用了函数 nativeWriteToParcel 来获取一个 Binder 本地对象，然后将这个对象写到 dest 对象中。函数 nativeWriteToParcel 在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static void nativeWriteToParcel(JNIEnv *env, jclass clazz, jint windowPtr,
  jobject parcelObj) {
    //获得关联的 C++层的 CursorWindow 对象 window
    CursorWindow *window = reinterpret_cast<CursorWindow*>(windowPtr);
    Parcel *parcel = parcelForJavaObject(env, parcelObj);

    status_t status = window->writeToParcel(parcel);
    if (status) {
        String8 msg;
        msg.appendFormat(
          "Could not write CursorWindow to Parcel due to error %d.", status);
        jniThrowRuntimeException(env, msg.string());
    }
}
```

在上述代码中，参数 parcelObj 指向 Java 层的 CursorWindow 对象。

9.3.4 处理进程通信的请求

函数 onTransact 的功能是处理类型为 IContentProvider.QUERY_TRANSACTION 的进程间的通信请求。函数 onTransact 在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中定义，具体实现代码如下所示：

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
  throws RemoteException {
    try {
        switch (code) {
            case QUERY_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);

                String callingPkg = data.readString();
                Uri url = Uri.CREATOR.createFromParcel(data);

                // String[] projection
                int num = data.readInt();
                String[] projection = null;
                if (num > 0) {
                    projection = new String[num];
                    for (int i=0; i<num; i++) {
                        projection[i] = data.readString();
                    }
                }

                // String selection, String[] selectionArgs...
                String selection = data.readString();
                num = data.readInt();
                String[] selectionArgs = null;
                if (num > 0) {
                    selectionArgs = new String[num];
                    for (int i=0; i<num; i++) {
                        selectionArgs[i] = data.readString();
                    }
                }

                String sortOrder = data.readString();
                IContentObserver observer =
                  IContentObserver.Stub.asInterface(data.readStrongBinder());
                ICancellationSignal cancellationSignal =
                  ICancellationSignal.Stub.asInterface(data.readStrongBinder());

                Cursor cursor = query(callingPkg, url, projection, selection,
                  selectionArgs, sortOrder, cancellationSignal);
```

```
        if (cursor != null) {
            try {
                CursorToBulkCursorAdaptor adaptor =
                  new CursorToBulkCursorAdaptor(
                  cursor, observer, getProviderName());
                BulkCursorDescriptor d = adaptor.getBulkCursorDescriptor();
                cursor = null;

                reply.writeNoException();
                reply.writeInt(1);
                d.writeToParcel(
                  reply, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
            } finally {
                // Close cursor if an exception was thrown
                // while constructing the adaptor.
                if (cursor != null) {
                    cursor.close();
                }
            }
        } else {
            reply.writeNoException();
            reply.writeInt(0);
        }

        return true;
    }

    case GET_TYPE_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        Uri url = Uri.CREATOR.createFromParcel(data);
        String type = getType(url);
        reply.writeNoException();
        reply.writeString(type);

        return true;
    }

    case INSERT_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        String callingPkg = data.readString();
        Uri url = Uri.CREATOR.createFromParcel(data);
        ContentValues values =
          ContentValues.CREATOR.createFromParcel(data);

        Uri out = insert(callingPkg, url, values);
        reply.writeNoException();
        Uri.writeToParcel(reply, out);
```

```
                return true;
            }

            case BULK_INSERT_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);
                String callingPkg = data.readString();
                Uri url = Uri.CREATOR.createFromParcel(data);
                ContentValues[] values =
                  data.createTypedArray (ContentValues.CREATOR);

                int count = bulkInsert(callingPkg, url, values);
                reply.writeNoException();
                reply.writeInt(count);
                return true;
            }

            case APPLY_BATCH_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);
                String callingPkg = data.readString();
                final int numOperations = data.readInt();
                final ArrayList<ContentProviderOperation> operations =
                  new ArrayList<ContentProviderOperation>(numOperations);
                for (int i=0; i<numOperations; i++) {
                    operations.add(i,
                      ContentProviderOperation.CREATOR.createFromParcel(data));
                }
                final ContentProviderResult[] results =
                  applyBatch(callingPkg, operations);
                reply.writeNoException();
                reply.writeTypedArray(results, 0);
                return true;
            }

            case DELETE_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);
                String callingPkg = data.readString();
                Uri url = Uri.CREATOR.createFromParcel(data);
                String selection = data.readString();
                String[] selectionArgs = data.readStringArray();

                int count = delete(callingPkg, url, selection, selectionArgs);

                reply.writeNoException();
                reply.writeInt(count);
                return true;
            }
```

```
case UPDATE_TRANSACTION:
{
    data.enforceInterface(IContentProvider.descriptor);
    String callingPkg = data.readString();
    Uri url = Uri.CREATOR.createFromParcel(data);
    ContentValues values =
      ContentValues.CREATOR.createFromParcel(data);
    String selection = data.readString();
    String[] selectionArgs = data.readStringArray();

    int count =
      update(callingPkg, url, values, selection, selectionArgs);

    reply.writeNoException();
    reply.writeInt(count);
    return true;
}

case OPEN_FILE_TRANSACTION:
{
    data.enforceInterface(IContentProvider.descriptor);
    String callingPkg = data.readString();
    Uri url = Uri.CREATOR.createFromParcel(data);
    String mode = data.readString();

    ParcelFileDescriptor fd;
    fd = openFile(callingPkg, url, mode);
    reply.writeNoException();
    if (fd != null) {
        reply.writeInt(1);
        fd.writeToParcel(reply,
          Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
    } else {
        reply.writeInt(0);
    }
    return true;
}

case OPEN_ASSET_FILE_TRANSACTION:
{
    data.enforceInterface(IContentProvider.descriptor);
    String callingPkg = data.readString();
    Uri url = Uri.CREATOR.createFromParcel(data);
    String mode = data.readString();

    AssetFileDescriptor fd;
    fd = openAssetFile(callingPkg, url, mode);
    reply.writeNoException();
```

```
        if (fd != null) {
            reply.writeInt(1);
            fd.writeToParcel(reply,
            Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            reply.writeInt(0);
        }
        return true;
    }

    case CALL_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);

        String callingPkg = data.readString();
        String method = data.readString();
        String stringArg = data.readString();
        Bundle args = data.readBundle();

        Bundle responseBundle = call(callingPkg, method, stringArg, args);

        reply.writeNoException();
        reply.writeBundle(responseBundle);
        return true;
    }

    case GET_STREAM_TYPES_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        Uri url = Uri.CREATOR.createFromParcel(data);
        String mimeTypeFilter = data.readString();
        String[] types = getStreamTypes(url, mimeTypeFilter);
        reply.writeNoException();
        reply.writeStringArray(types);

        return true;
    }

    case OPEN_TYPED_ASSET_FILE_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        String callingPkg = data.readString();
        Uri url = Uri.CREATOR.createFromParcel(data);
        String mimeType = data.readString();
        Bundle opts = data.readBundle();

        AssetFileDescriptor fd;
        fd = openTypedAssetFile(callingPkg, url, mimeType, opts);
        reply.writeNoException();
```

```
                if (fd != null) {
                    reply.writeInt(1);
                    fd.writeToParcel(reply,
                      Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
                } else {
                    reply.writeInt(0);
                }
                return true;
            }

            case CREATE_CANCELATION_SIGNAL_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);

                ICancellationSignal cancellationSignal =
                  createCancellationSignal();
                reply.writeNoException();
                reply.writeStrongBinder(cancellationSignal.asBinder());
                return true;
            }
        }
    } catch (Exception e) {
        DatabaseUtils.writeExceptionToParcel(reply, e);
        return true;
    }
    return super.onTransact(code, data, reply, flags);
}
```

在上述代码中，函数 createFromParcel 从数据流 data 中重建一个本地的 CursorWindow 对象，然后将数据流 data 的下一个整数值读取出来。如果这个整数值不为 0，则变量 wantsCursorMetadata 的值为 true，表示 Content Provider 在返回 IBulkCursor 接口给第三方应用程序之前要先执行一次数据库查询操作，就可以把结果数据的元信息返回给第三方应用程序。

下面来看函数 createFromParcel，功能是将倒数第二个进程间的通信数据封装成一个 CursorWindow 对象。函数 createFromParcel 在文件 frameworks/base/core/java/android/content/CursorWindow.java 中定义，具体实现代码如下所示：

```
public static final Parcelable.Creator<CursorWindow> CREATOR=
  new Parcelable.Creator<CursorWindow>() {
    public CursorWindow createFromParcel(Parcel source) {
        return new CursorWindow(source);
    }

    public CursorWindow[] newArray(int size) {
        return new CursorWindow[size];
    }
};
```

在上述代码中，使用参数 source 创建了一个 CursorWindow。

创建 CursorWindow 对象的功能是通过函数 CursorWindow 实现的，此函数在文件 frameworks/base/core/java/android/content/CursorWindow.java 中定义，具体实现代码如下所示：

```
private CursorWindow(Parcel source) {
    mStartPos = source.readInt();
    mWindowPtr = nativeCreateFromParcel(source);
    if (mWindowPtr == 0) {
        throw new CursorWindowAllocationException("Cursor window could not be "
          + "created from binder.");
    }
    mName = nativeGetName(mWindowPtr);
    mCloseGuard.open("close");
}
```

在上述创建 CursorWindow 对象的过程中，首先是从数据流 source 中将写入的 Binder 接口读取出来，然后使用这个 Binder 接口初始化这个 CursorWindow 对象。其实这个 Binder 接口的实际类型为 IMemory，在里面封装了对匿名共享内存的访问操作。初始化这个匿名共享内存对象的操作是由本地函数 nativeGetName 实现的，此函数在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static jstring nativeGetName(JNIEnv *env, jclass clazz, jint windowPtr) {
    CursorWindow *window = reinterpret_cast<CursorWindow*>(windowPtr);
    return env->NewStringUTF(window->name().string());
}
```

这样便分别在 Java 层和 C++层将 CursorWindow 对象实现了关联。

9.3.5 数据操作

再看文件 frameworks/base/core/java/android/database/sqlite/SQLiteQueryBuilder.java 中的函数 query，功能是准备一个数据库查询计划，此函数的具体实现代码如下所示：

```
public Cursor query(SQLiteDatabase db, String[] projectionIn,
  String selection, String[] selectionArgs, String groupBy,
  String having, String sortOrder, String limit,
  CancellationSignal cancellationSignal) {
    if (mTables == null) {
        return null;
    }

    if (mStrict && selection!=null && selection.length()>0) {
        String sqlForValidation = buildQuery(projectionIn, "(" + selection + ")",
          groupBy, having, sortOrder, limit);
        validateQuerySql(db, sqlForValidation,
          cancellationSignal); // will throw if query is invalid
    }
    String sql = buildQuery(
      projectionIn, selection, groupBy, having, sortOrder, limit);
    if (Log.isLoggable(TAG, Log.DEBUG)) {
```

```
        Log.d(TAG, "Performing query: " + sql);
    }
    return db.rawQueryWithFactory(mFactory, sql, selectionArgs,
      SQLiteDatabase.findEditTable(mTables),
      cancellationSignal); // will throw if query is invalid
}
```

在上述代码中，调用函数 buildQuery 构造了一个 SQL 语句，能够根据从参数传来的列名子句、select 子句、where 子句、group by 子句、having 子句、order 子句以及 limit 子句构造一个完整的 SQL 语句。构造完成这个 SQL 查询语句后，调用从参数传下来的数据库对象 db 的函数 rawQueryWithFactory 来实现进一步操作。函数 rawQueryWithFactory 在文件 frameworks/base/core/java/android/database/sqlite/SQLiteDatabase.java 中定义，具体实现代码如下所示：

```
public Cursor rawQueryWithFactory(
CursorFactory cursorFactory, String sql, String[] selectionArgs,
String editTable, CancellationSignal cancellationSignal) {
    acquireReference();
    try {
        SQLiteCursorDriver driver = new SQLiteDirectCursorDriver(
          this, sql, editTable, cancellationSignal);
        return driver.query(
          cursorFactory!=null? cursorFactory : mCursorFactory, selectionArgs);
    } finally {
        releaseReference();
    }
}
```

在上述代码中，创建了一个 SQLiteCursorDriver 对象 driver，然后调用它的成员函数 query 创建了一个 Cursor 对象，此 Cursor 对象的类型是 SQLiteCursor。来看 query 函数的代码：

```
public Cursor query(CursorFactory factory, String[] selectionArgs) {
    final SQLiteQuery query =
      new SQLiteQuery(mDatabase, mSql, mCancellationSignal);
    final Cursor cursor;
    try {
        query.bindAllArgsAsStrings(selectionArgs);

        if (factory == null) {
            cursor = new SQLiteCursor(this, mEditTable, query);
        } else {
            cursor = factory.newCursor(mDatabase, this, mEditTable, query);
        }
    } catch (RuntimeException ex) {
        query.close();
        throw ex;
    }
    mQuery = query;
    return cursor;
}
```

在上述代码中，会先根据数据库对象 mDatabase 和原生 SQL 语句构造一个 SQLiteQuery 对象。在创建这个对象的过程中，会解析这个原生 SQL 语句，并创建数据库查询计划，这样在等到真正查询时，就可以马上从数据库中获得取数据，而无须分析和理解这个 SQL 字符串语句，上述整个过程被称为 SQL 语句编译。有了这个 SQLiteQuery 对象之后，再将其与数据库对象 mDatabase 等待信息一起来创建一个 SQLiteCursor 对象，这样这个 SQLiteCursor 对象就可以圈定将来要从数据库中获取的数据了。当执行完这一步骤之后，就可以把这个 SQLiteCursor 对象返回给上层，最终返回到类 Transport 中的函数 bulkQuery 中。有了这个 SQLiteCursor 对象后，就可以通过创建一个 CursorToBulkCursorAdaptor 对象的方式把它与匿名共享内存关联起来，这样就为将来从数据库中查询得到的数据找到了归宿。CursorToBulkCursorAdaptor 对象的实现文件是 frameworks/base/core/java/android/database/CursorToBulkCursorAdaptor.java，其成员函数 CursorToBulkCursorAdaptor 的功能是将 SQLiteCurosr 对象转换为一个 AbstractWindowedCursor 对象，目的是为了调用函数 setWindow 把传进来的 CursorWindow 对象 window 保存起来，以便后面用来保存数据。函数 CursorToBulkCursorAdaptor 的具体实现代码如下所示：

```
public CursorToBulkCursorAdaptor(Cursor cursor, IContentObserver observer,
  String providerName) {
    if (cursor instanceof CrossProcessCursor) {
        mCursor = (CrossProcessCursor)cursor;
    } else {
        mCursor = new CrossProcessCursorWrapper(cursor);
    }
    mProviderName = providerName;
    synchronized (mLock) {
        createAndRegisterObserverProxyLocked(observer);
    }
}
```

然后执行类 SQLiteCursor 中的成员函数 getCount，功能是获取 MainActivity 组件获取信息的请求。函数 getCount 在文件 frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java 中定义，具体实现代码如下所示：

```
public int getCount() {
    if (mCount == NO_COUNT) {
        fillWindow(0);
    }
    return mCount;
}
```

在上述代码中，变量 mCount 的初始化值为 NO_COUNT，表示还没有去执行数据库查询操作，所以不知道它的值是多少，这需要通过调用函数 fillWindow 从数据库中查询获得。函数 fillWindow 在文件 frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java 中定义，具体实现代码如下所示：

```
private void fillWindow(int requiredPos) {
    clearOrCreateWindow(getDatabase().getPath());
```

```
    try {
        if (mCount == NO_COUNT) {
            int startPos =
              DatabaseUtils.cursorPickFillWindowStartPosition(requiredPos, 0);
            mCount = mQuery.fillWindow(mWindow, startPos, requiredPos, true);
            mCursorWindowCapacity = mWindow.getNumRows();
            if (Log.isLoggable(TAG, Log.DEBUG)) {
                Log.d(TAG, "received count(*) from native_fill_window: " + mCount);
            }
        } else {
            int startPos =
              DatabaseUtils.cursorPickFillWindowStartPosition(
                requiredPos, mCursorWindowCapacity);
            mQuery.fillWindow(mWindow, startPos, requiredPos, false);
        }
    } catch (RuntimeException ex) {
        closeWindow();
        throw ex;
    }
}
```

到此为止，最终就可以把从 Content Provider 中查询得到的数据通过匿名共享内存返回给第三方应用程序了。这样就完成了 Android 应用程序组件 Content Provider 在应用程序之间共享数据的原理分析。

由此可见，整个过程是通过 Binder 进程间的通信机制和匿名共享内存来实现的。

Android

第 10 章 Broadcast(广播)系统详解

在 Android 系统中，Broadcast 是一种广泛运用的在应用程序之间传输信息的机制。而 BroadcastReceiver 用于接收广播信息，是对发送出来的 Broadcast 进行过滤接收并响应的一类组件。本章将详细讲解 Android 4.3 中 Broadcast(广播)系统的源码，为读者步入本书后面高级知识的学习打下基础。

10.1 Broadcast基础

(1) 在 Android 系统中，发送 Broadcast 和使用 BroadcastReceiver 过滤接收的过程如下。

① 首先在需要发送信息的地方，把要发送的信息和用于过滤的信息(如 Action、Category)装入一个 Intent 对象。

② 通过调用 Context.sendBroadcast()、sendOrderBroadcast()或 sendStickyBroadcast()方法，把 Intent 对象以广播方式发送出去。

③ 当发送 Intent 以后，所有已经注册的 BroadcastReceiver 会检查注册时的 IntentFilter 是否与发送的 Intent 相匹配，如果匹配，就会调用 BroadcastReceiver 的 onReceive()方法。所以当我们定义一个 BroadcastReceiver 的时候，都需要实现 onReceive()方法。

在现实开发应用中，有如下两种注册 BroadcastReceiver 的方式。

- 静态方式：在文件 AndroidManifest.xml 中用<receiver>标签声明注册，并在标签内用<intent-filter>标签设置过滤器。
- 动态方式：在代码中先定义并设置好一个 IntentFilter 对象，然后在需要注册的地方调用 Context.registerReceiver()方法，要取消时，就调用 Context.unregisterReceiver()方法。当用动态方式注册的 BroadcastReceiver 的 Context 对象被销毁时，BroadcastReceiver 也就自动取消注册了(特别注意，有些可能是需要后台监听的，如短信消息)。

如果在使用 sendBroadcast()的方法时指定了接收权限，则只有在 AndroidManifest.xml 中用<uses-permission>标签声明了拥有此权限的 BroascastReceiver 才会有可能接收到发送来的 Broadcast。同样，若在注册 BroadcastReceiver 时指定了可接收的 Broadcast 的权限，则只有在包内的 AndroidManifest.xml 中用<uses-permission>标签声明了，拥有此权限的 Context 对象所发送的 Broadcast 才能被这个 BroadcastReceiver 所接收。

(2) 在 Android 开发应用中，广播事件的基本流程如下所示。

① 注册广播事件：注册方式有两种，一种是静态注册，就是在 AndroidManifest.xml 文件中定义，注册的广播接收器必须要继承 BroadcastReceiver；另一种是动态注册，是在程序中使用 Context.registerReceiver 注册的，注册的广播接收器相当于一个匿名类。这两种方式都需要 IntentFilter。

② 发送广播事件：通过 Context.sendBroadcast 发送，由 Intent 传递注册时用到的 Action。

③ 接收广播事件：当发送的广播被接收器监听到后，会调用它的 onReceive()方法，并将包含消息的 Intent 对象传给它。onReceive 方法中代码的执行时间不要超过 5s，否则 Android 会弹出超时对话框。

10.2 发送广播信息

在 Android 系统中，发送广播功能是通过 sendBroadcast 实现的，整个过程以 ActivityManagerService 为中心。广播的发送者将广播发送到 ActivityManagerService，当

ActivityManagerService 接收到这个广播后，会在自己的注册中心查看有哪些广播接收器订阅了这个广播，然后将此广播逐一发送到这些广播接收器中。上述广播过程中的发送和处理是异步实现的，ActivityManagerService 并不等待广播接收器处理完这些广播就会返回。由此可见，广播的发送路径就是从发送者到 ActivityManagerService，再从 ActivityManagerService 到接收者，这中间的两个过程都是通过 Binder 进程间通信机制来完成的。

在本节的内容中，将详细分析 Android 4.3 中发送广播信息的源码。

10.2.1　intent描述指示

在 Android 应用开发过程中，当在某个 service 中想要发送广播时，通常会调用如下代码来实现：

```
Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
intent.putExtra(COUNTER_VALUE, counter);
sendBroadcast(intent);
```

Android 中的广播是使用 Intent 来描述的，BROADCAST_COUNTER_ACTION 名称就是用来与广播接收者的类型进行匹配的。在类 Intent 中的定义代码如下所示：

```
public class Intent implements Parcelable, Cloneable {
    // ---------------------------------------------------------------------
    private String mAction;
    private Uri mData;
    private String mType;
    private String mPackage;
    private ComponentName mComponent;
    private int mFlags;
    private HashSet<String> mCategories;
    private Bundle mExtras;
    private Rect mSourceBounds;
}
```

在上述代码中，变量 mAction 和 mExtras 不为空，其余为空。

10.2.2　传递广播信息

文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义了 sendBroadcast 函数，功能是调用 ContextImpl.sendBroadcast 实现进一步的操作，具体实现代码如下所示：

```
public class ContextWrapper extends Context {
    Context mBase;

    public ContextWrapper(Context base) {
        mBase = base;
    }

    @Override
    public void sendBroadcast(Intent intent) {
```

```
        mBase.sendBroadcast(intent);
    }
    ...
}
```

在上述代码中，变量 mBase 是一个 ContextImpl 实例。

再看文件 frameworks/base/core/java/android/app/ContextImpl.java 中的函数 sendBroadcast，功能是调用类 ActivityManagerService 中的远程接口 ActivityManagerProxy，将这个广播信息发送给 ActivityManagerService。函数 sendBroadcast 的具体实现代码如下所示：

```
public void sendBroadcast(Intent intent) {
    warnIfCallingFromSystemProcess();
    String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
    try {
        intent.prepareToLeaveProcess();
        ActivityManagerNative.getDefault().broadcastIntent(
          mMainThread.getApplicationThread(), intent, resolvedType, null,
          Activity.RESULT_OK,null,null,null,AppOpsManager.OP_NONE,false,false,
          getUserId());
    } catch (RemoteException e) {}
}
```

在上述代码中，resolvedType 表示这个 Intent 的 MIME 类型。

10.2.3 封装传递

再看 frameworks/base/core/java/android/app/ActivityManagerNative.java 中的 broadcastIntent 函数，功能是封装传递的参数，并通过 Binder 驱动程序进入到类 ActivityManagerService 中的函数 broadcastIntent 中。函数 broadcastIntent 的具体实现代码如下所示：

```
public int broadcastIntent(IApplicationThread caller,
  Intent intent, String resolvedType,  IIntentReceiver resultTo,
  int resultCode, String resultData, Bundle map,
  String requiredPermission, int appOp, boolean serialized,
  boolean sticky, int userId) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    //将传进来的参数写入 data 对象中
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    intent.writeToParcel(data, 0);
    data.writeString(resolvedType);
    data.writeStrongBinder(resultTo!=null? resultTo.asBinder() : null);
    data.writeInt(resultCode);
    data.writeString(resultData);
    data.writeBundle(map);
    data.writeString(requiredPermission);
    data.writeInt(appOp);
    data.writeInt(serialized? 1 : 0);
```

```
    data.writeInt(sticky? 1 : 0);
    data.writeInt(userId);
    mRemote.transact(BROADCAST_INTENT_TRANSACTION, data, reply, 0);
    reply.readException();
    int res = reply.readInt();
    reply.recycle();
    data.recycle();
    return res;
}
```

10.2.4　处理发送请求

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的 broadcastIntent 函数，功能是处理类型为 BROADCAST_INTENT_TRANSACTION 的进程通信请求。函数 broadcastIntent 的具体实现代码如下所示：

```
public final int broadcastIntent(IApplicationThread caller,
  Intent intent, String resolvedType, IIntentReceiver resultTo,
  int resultCode, String resultData, Bundle map,
  String requiredPermission, int appOp, boolean serialized,
  boolean sticky, int userId) {
    enforceNotIsolatedCaller("broadcastIntent");
    synchronized(this) {
        //验证 intent 描述的广播内容是否合法
        intent = verifyBroadcastLocked(intent);
        //获取发送广播进程的身份
        final ProcessRecord callerApp = getRecordForAppLocked(caller);
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        //调用函数 broadcastIntentLocked 处理参数 intent 描述的广播
        int res = broadcastIntentLocked(callerApp,
          callerApp!=null? callerApp.info.packageName : null,
          intent, resolvedType, resultTo,
          resultCode, resultData, map, requiredPermission, appOp,
          serialized, sticky, callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }
}
```

10.2.5　查找广播接收者

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的 broadcastIntentLocked 函数，功能是查找目标广播的接收者。此函数先根据 Intent 找出相应的广播接收器，然后验证是否设置了这个 Intent 的 Intent.FLAG_RECEIVER_REPLACE_

PENDING 位，如果没有，则 ActivityManagerService 会在当前的系统中查看有没有相同的 Intent 还未被处理，如果有，则用当前新的 Intent 来替换旧的 Intent。函数 broadcastIntentLocked 的具体实现代码如下所示：

```
private final int broadcastIntentLocked(ProcessRecord callerApp,
  String callerPackage, Intent intent, String resolvedType,
  IIntentReceiver resultTo, int resultCode, String resultData,
  Bundle map, String requiredPermission, int appOp,
  boolean ordered, boolean sticky, int callingPid, int callingUid,
  int userId) {
    intent = new Intent(intent);

    // By default broadcasts do not go to stopped apps.
    intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);

    if (DEBUG_BROADCAST_LIGHT)
        Slog.v(
          TAG, (sticky ? "Broadcast sticky: ": "Broadcast: ") + intent
          + " ordered=" + ordered + " userid=" + userId);
    if ((resultTo != null) && !ordered) {
        Slog.w(TAG, "Broadcast " + intent
          + " not ordered but result callback requested!");
    }
    ...
    // 根据 intent 找出相应的广播接收器
    List receivers = null;
    List<BroadcastFilter> registeredReceivers = null;
    //Need to resolve the intent to interested receivers...
    if ((intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY) == 0) {
        receivers = collectReceiverComponents(intent, resolvedType, users);
    }
    if (intent.getComponent() == null) {
        registeredReceivers =
          mReceiverResolver.queryIntent(intent, resolvedType, false, userId);
    }
    //验证是否设置 intent 的 Intent.FLAG_RECEIVER_REPLACE_PENDING 位
    final boolean replacePending =
      (intent.getFlags()&Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0;

    if (DEBUG_BROADCAST)
        Slog.v(TAG, "Enqueing broadcast: " + intent.getAction()
          + " replacePending=" + replacePending);

    int NR = registeredReceivers!=null? registeredReceivers.size() : 0;
    if (!ordered && NR > 0) {
        // If we are not serializing this broadcast, then send the
        // registered receivers separately so they don't wait for the
        // components to be launched.
```

```
        final BroadcastQueue queue = broadcastQueueForIntent(intent);
        BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
          callerPackage, callingPid, callingUid, requiredPermission, appOp,
          registeredReceivers, resultTo, resultCode, resultData, map,
          ordered, sticky, false, userId);
        if (DEBUG_BROADCAST)
            Slog.v(TAG, "Enqueueing parallel broadcast " + r);
            final boolean replaced =
              replacePending && queue.replaceParallelBroadcastLocked(r);
        if (!replaced) {
            queue.enqueueParallelBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();
        }
        registeredReceivers = null;
        NR = 0;
    }

    // Merge into one list.
    int ir = 0;
    if (receivers != null) {
        // A special case for PACKAGE_ADDED: do not allow the package
        // being added to see this broadcast.  This prevents them from
        // using this as a back door to get run as soon as they are
        // installed.  Maybe in the future we want to have a special install
        // broadcast or such for apps, but we'd like to deliberately make
        // this decision.
        String skipPackages[] = null;
        if (Intent.ACTION_PACKAGE_ADDED.equals(intent.getAction())
          || Intent.ACTION_PACKAGE_RESTARTED.equals(intent.getAction())
          || Intent.ACTION_PACKAGE_DATA_CLEARED.equals(intent.getAction())) {
            Uri data = intent.getData();
            if (data != null) {
                String pkgName = data.getSchemeSpecificPart();
                if (pkgName != null) {
                    skipPackages = new String[] { pkgName };
                }
            }
        } else if (Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE
          .equals (intent.getAction())) {
            skipPackages =
              intent.getStringArrayExtra(Intent.EXTRA_CHANGED_PACKAGE_LIST);
        }
        if (skipPackages!=null && (skipPackages.length>0)) {
            //循环验证是否存在和参数 Intent 一样的广播
            for (String skipPackage : skipPackages) {
                if (skipPackage != null) {
                    int NT = receivers.size();
                    for (int it=0; it<NT; it++) {
                        ResolveInfo curt = (ResolveInfo)receivers.get(it);
```

```
                    if (curt.activityInfo.packageName.equals(skipPackage)) {
                        receivers.remove(it);
                        it--;
                        NT--;
                    }
                }
            }
        }
    }
    int NT = receivers!=null? receivers.size() : 0;
    int it = 0;
    ResolveInfo curt = null;
    BroadcastFilter curr = null;
    while (it<NT && ir<NR) {
        if (curt == null) {
            curt = (ResolveInfo)receivers.get(it);
        }
        if (curr == null) {
            curr = registeredReceivers.get(ir);
        }
        if (curr.getPriority() >= curt.priority) {
            // Insert this broadcast record into the final list.
            receivers.add(it, curr);
            ir++;
            curr = null;
            it++;
            NT++;
        } else {
            // Skip to the next ResolveInfo in the final list.
            it++;
            curt = null;
        }
    }
}
while (ir < NR) {
    if (receivers == null) {
        receivers = new ArrayList();
    }
    receivers.add(registeredReceivers.get(ir));
    ir++;
}

if ((receivers != null && receivers.size() > 0)
  || resultTo != null) {
    BroadcastQueue queue = broadcastQueueForIntent(intent);
    BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
      callerPackage, callingPid, callingUid, requiredPermission, appOp,
      receivers, resultTo, resultCode, resultData, map, ordered,
      sticky, false, userId);
```

```
        if (DEBUG_BROADCAST)
            Slog.v(TAG, "Enqueueing ordered broadcast " + r
              + ": prev had " + queue.mOrderedBroadcasts.size());
        if (DEBUG_BROADCAST) {
            int seq = r.intent.getIntExtra("seq", -1);
            Slog.i(TAG, "Enqueueing broadcast "
              + r.intent.getAction() + " seq=" + seq);
        }
        boolean replaced =
          replacePending && queue.replaceOrderedBroadcastLocked(r);
        if (!replaced) {
            queue.enqueueOrderedBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();
        }
    }
    return ActivityManager.BROADCAST_SUCCESS;
}
```

在上述代码中，成员变量 mHandler 在类 ActivityManagerService 的内部被定义，是一个 Handler 类变量，通过此类中的函数 sendEmptyMessage，可以将一个类型为 BROADCAST_INTENT_MSG 的空消息放进 ActivityManagerService 的消息队列中去。此处的空消息是指这个消息除了有类型信息之外，没有任何其他额外的信息。

10.2.6　处理广播信息

再看函数 scheduleBroadcastsLocked，具体实现代码如下所示：

```
private final void scheduleBroadcastsLocked() {
    if (DEBUG_BROADCAST)
        Slog.v(TAG, "Schedule broadcasts: current=" + mBroadcastsScheduled);
    if (mBroadcastsScheduled) {
        return;
    }
    mHandler.sendEmptyMessage(BROADCAST_INTENT_MSG);
    mBroadcastsScheduled = true;
}
```

在上述代码中，mBroadcastsScheduled 表示是否已经向所有运行的线程发送了一个类型为 BROADCAST_INTENT_MSG 的消息。

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 handleMessage，功能是处理类型为 BROADCAST_INTENT_MSG 的广播信息。主要实现代码如下所示：

```
public void handleMessage(Message msg) {
    switch (msg.what) {
        case SHOW_ERROR_MSG: {
            HashMap data = (HashMap)msg.obj;
            boolean showBackground =
```

```
              Settings.Secure.getInt(mContext.getContentResolver(),
              Settings.Secure.ANR_SHOW_BACKGROUND, 0)!=0;
            synchronized(ActivityManagerService.this) {
                ProcessRecord proc = (ProcessRecord)data.get("app");
                AppErrorResult res = (AppErrorResult)data.get("result");
                if (proc!=null && proc.crashDialog!=null) {
                    Slog.e(TAG, "App already has crash dialog: " + proc);
                    if (res != null) {
                        res.set(0);
                    }
                    return;
                }
                if (!showBackground
                  && UserHandle.getAppId(proc.uid)>=Process.FIRST_APPLICATION_UID
                  && proc.userId!=mCurrentUserId && proc.pid!=MY_PID) {
                    Slog.w(TAG, "Skipping crash dialog of " + proc + ": background");
                    if (res != null) {
                        res.set(0);
                    }
                    return;
                }
                if (mShowDialogs && !mSleeping && !mShuttingDown) {
                    Dialog d = new AppErrorDialog(mContext,
                      ActivityManagerService.this, res, proc);
                    d.show();
                    proc.crashDialog = d;
                } else {
                    // The device is asleep, so just pretend that the user
                    // saw a crash dialog and hit "force quit".
                    if (res != null) {
                        res.set(0);
                    }
                }
            }
            ensureBootCompleted();
        } break;
        case BROADCAST_INTENT_MSG: {
            ...
            //调用函数processNextBroadcast来处理下一个未处理的广播
            processNextBroadcast(true);
        } break;
        ....
    }
}
```

在上述代码中，调用类ActivityManagerService中的函数processNextBroadcast来处理下一个未处理的广播。函数processNextBroadcast在文件frameworks/base/services/java/com/android/server/am/BroadcastQueue.java中定义，具体实现代码如下所示：

```
final void processNextBroadcast(boolean fromMsg) {
    synchronized(mService) {
        BroadcastRecord r;

        if (DEBUG_BROADCAST)
            Slog.v(TAG, "processNextBroadcast ["
                + mQueueName + "]: "
                + mParallelBroadcasts.size() + " broadcasts, "
                + mOrderedBroadcasts.size() + " ordered broadcasts");

        mService.updateCpuStats();

        if (fromMsg) {
            mBroadcastsScheduled = false;
        }

        // First, deliver any non-serialized broadcasts right away.
        while (mParallelBroadcasts.size() > 0) {
            r = mParallelBroadcasts.remove(0);
            r.dispatchTime = SystemClock.uptimeMillis();
            r.dispatchClockTime = System.currentTimeMillis();
            final int N = r.receivers.size();
            if (DEBUG_BROADCAST_LIGHT)
                Slog.v(TAG, "Processing parallel broadcast ["
                  + mQueueName + "] " + r);
            for (int i=0; i<N; i++) {
                Object target = r.receivers.get(i);
                if (DEBUG_BROADCAST)
                    Slog.v(TAG, "Delivering non-ordered on [" + mQueueName
                      + "] to registered " + target + ": " + r);
                deliverToRegisteredReceiverLocked(
                  r, (BroadcastFilter)target, false);
            }
            addBroadcastToHistoryLocked(r);
            if (DEBUG_BROADCAST_LIGHT)
                Slog.v(TAG, "Done with parallel broadcast ["
                  + mQueueName + "] " + r);
        }
        if (mPendingBroadcast != null) {
            if (DEBUG_BROADCAST_LIGHT) {
                Slog.v(TAG, "processNextBroadcast ["
                  + mQueueName + "]: waiting for "
                  + mPendingBroadcast.curApp);
            }

            boolean isDead;
            synchronized(mService.mPidsSelfLocked) {
                isDead = (mService.mPidsSelfLocked.get(
                  mPendingBroadcast.curApp.pid) == null);
```

```
            }
            if (!isDead) {
                // It's still alive, so keep waiting
                return;
            } else {
                Slog.w(TAG, "pending app  ["
                  + mQueueName + "]" + mPendingBroadcast.curApp
                  + " died before responding to broadcast");
                mPendingBroadcast.state = BroadcastRecord.IDLE;
                mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
                mPendingBroadcast = null;
            }
        }

        boolean looped = false;

        do {
            if (mOrderedBroadcasts.size() == 0) {
                // No more broadcasts pending, so all done!
                mService.scheduleAppGcsLocked();
                if (looped) {
                    mService.updateOomAdjLocked();
                }
                return;
            }
            r = mOrderedBroadcasts.get(0);
            boolean forceReceive = false;
            int numReceivers = (r.receivers!=null)? r.receivers.size() : 0;
            if (mService.mProcessesReady && r.dispatchTime > 0) {
                long now = SystemClock.uptimeMillis();
                if ((numReceivers > 0)
                  && (now > r.dispatchTime + (2*mTimeoutPeriod*numReceivers))) {
                    Slog.w(TAG, "Hung broadcast ["
                      + mQueueName + "] discarded after timeout failure:"
                      + " now=" + now
                      + " dispatchTime=" + r.dispatchTime
                      + " startTime=" + r.receiverTime
                      + " intent=" + r.intent
                      + " numReceivers=" + numReceivers
                      + " nextReceiver=" + r.nextReceiver
                      + " state=" + r.state);
                    broadcastTimeoutLocked(false); // forcibly finish this broadcast
                    forceReceive = true;
                    r.state = BroadcastRecord.IDLE;
                }
            }

            if (r.state != BroadcastRecord.IDLE) {
                if (DEBUG_BROADCAST)
```

```
                Slog.d(TAG, "processNextBroadcast("
                  + mQueueName + ") called when not idle (state="
                  + r.state + ")");
              return;
          }

          if (r.receivers == null || r.nextReceiver >= numReceivers
            || r.resultAbort || forceReceive) {
              if (r.resultTo != null) {
                  try {
                      if (DEBUG_BROADCAST) {
                          int seq = r.intent.getIntExtra("seq", -1);
                          Slog.i(TAG, "Finishing broadcast ["
                            + mQueueName + "] " + r.intent.getAction()
                            + " seq=" + seq + " app=" + r.callerApp);
                      }
                      performReceiveLocked(r.callerApp, r.resultTo,
                        new Intent(r.intent), r.resultCode,
                        r.resultData, r.resultExtras, false, false, r.userId);
                      r.resultTo = null;
                  } catch (RemoteException e) {
                      Slog.w(TAG, "Failure ["
                        + mQueueName + "] sending broadcast result of "
                        + r.intent, e);
                  }
              }

              if (DEBUG_BROADCAST)
                Slog.v(TAG, "Cancelling BROADCAST TIMEOUT MSG");
              cancelBroadcastTimeoutLocked();

              if (DEBUG_BROADCAST_LIGHT)
                Slog.v(TAG, "Finished with ordered + broadcast " r);
              addBroadcastToHistoryLocked(r);
              mOrderedBroadcasts.remove(0);
              r = null;
              looped = true;
              continue;
          }
      } while (r == null);

      // Get the next receiver...
      int recIdx = r.nextReceiver++;
      r.receiverTime = SystemClock.uptimeMillis();
      if (recIdx == 0) {
          r.dispatchTime = r.receiverTime;
          r.dispatchClockTime = System.currentTimeMillis();
          if (DEBUG_BROADCAST_LIGHT)
            Slog.v(TAG, "Processing ordered broadcast ["
```

```
                + mQueueName + "] " + r);
        }
        if (! mPendingBroadcastTimeoutMessage) {
            long timeoutTime = r.receiverTime + mTimeoutPeriod;
            if (DEBUG_BROADCAST)
                Slog.v(TAG, "Submitting BROADCAST_TIMEOUT_MSG ["
                  + mQueueName + "] for " + r + " at " + timeoutTime);
            setBroadcastTimeoutLocked(timeoutTime);
        }

        Object nextReceiver = r.receivers.get(recIdx);
        if (nextReceiver instanceof BroadcastFilter) {
            BroadcastFilter filter = (BroadcastFilter)nextReceiver;
            if (DEBUG_BROADCAST)
                Slog.v(TAG, "Delivering ordered ["
                  + mQueueName + "] to registered " + filter + ": " + r);
            deliverToRegisteredReceiverLocked(r, filter, r.ordered);
            if (r.receiver == null || !r.ordered) {
                if (DEBUG_BROADCAST)
                    Slog.v(TAG, "Quick finishing ["
                      + mQueueName + "]: ordered="
                      + r.ordered + " receiver=" + r.receiver);
                r.state = BroadcastRecord.IDLE;
                scheduleBroadcastsLocked();
            }
            return;
        }
        ResolveInfo info = (ResolveInfo)nextReceiver;
        ComponentName component = new ComponentName(
          info.activityInfo.applicationInfo.packageName,
          info.activityInfo.name);

        boolean skip = false;
        int perm = mService.checkComponentPermission(
          info.activityInfo.permission, r.callingPid, r.callingUid,
          info.activityInfo.applicationInfo.uid, info.activityInfo.exported);
        if (perm != PackageManager.PERMISSION_GRANTED) {
            if (!info.activityInfo.exported) {
                Slog.w(TAG, "Permission Denial: broadcasting "
                  + r.intent.toString()
                  + " from " + r.callerPackage + " (pid=" + r.callingPid
                  + ", uid=" + r.callingUid + ")" + " is not exported from uid "
                  + info.activityInfo.applicationInfo.uid
                  + " due to receiver " + component.flattenToShortString());
            } else {
                Slog.w(TAG, "Permission Denial: broadcasting "
                  + r.intent.toString()
                  + " from " + r.callerPackage + " (pid=" + r.callingPid
                  + ", uid=" + r.callingUid + ")"
```

```
                + " requires " + info.activityInfo.permission
                + " due to receiver " + component.flattenToShortString());
        }
        skip = true;
    }
    if (info.activityInfo.applicationInfo.uid != Process.SYSTEM_UID
      && r.requiredPermission != null) {
        try {
            perm = AppGlobals
              .getPackageManager().checkPermission(r.requiredPermission,
                info.activityInfo.applicationInfo.packageName);
        } catch (RemoteException e) {
            perm = PackageManager.PERMISSION_DENIED;
        }
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: receiving "
              + r.intent + " to "
              + component.flattenToShortString()
              + " requires " + r.requiredPermission
              + " due to sender " + r.callerPackage
              + " (uid " + r.callingUid + ")");
            skip = true;
        }
    }
    if (r.appOp != AppOpsManager.OP_NONE) {
        int mode = mService.mAppOpsService.checkOperation(
          r.appOp, info.activityInfo.applicationInfo.uid,
          info.activityInfo.packageName);
        if (mode != AppOpsManager.MODE_ALLOWED) {
            if (DEBUG_BROADCAST)
                Slog.v(TAG, "App op " + r.appOp
            + " not allowed for broadcast to uid "
            + info.activityInfo.applicationInfo.uid + " pkg "
            + info.activityInfo.packageName);
            skip = true;
        }
    }
    boolean isSingleton = false;
    try {
        isSingleton = mService.isSingleton(info.activityInfo.processName,
          info.activityInfo.applicationInfo,
          info.activityInfo.name, info.activityInfo.flags);
    } catch (SecurityException e) {
        Slog.w(TAG, e.getMessage());
        skip = true;
    }
    if ((info.activityInfo.flags&ActivityInfo.FLAG_SINGLE_USER) != 0) {
        if (ActivityManager.checkUidPermission(
          android.Manifest.permission.INTERACT_ACROSS_USERS,
```

```
                info.activityInfo.applicationInfo.uid)
                != PackageManager.PERMISSION_GRANTED) {
                    Slog.w(TAG, "Permission Denial: Receiver "
                      + component.flattenToShortString()
                    + " requests FLAG_SINGLE_USER, but app does not hold "
                    + android.Manifest.permission.INTERACT_ACROSS_USERS);
                skip = true;
            }
        }
        if (r.curApp != null && r.curApp.crashing) {
            if (DEBUG_BROADCAST)  Slog.v(TAG,
                "Skipping deliver ordered ["
                + mQueueName + "] " + r + " to " + r.curApp
                + ": process crashing");
                skip = true;
            }

            if (skip) {
                if (DEBUG_BROADCAST)
                  Slog.v(TAG, "Skipping delivery of ordered ["
                    + mQueueName + "] " + r + " for whatever reason");
                    r.receiver = null;
                    r.curFilter = null;
                    r.state = BroadcastRecord.IDLE;
                    scheduleBroadcastsLocked();
                    return;
            }

            r.state = BroadcastRecord.APP_RECEIVE;
            String targetProcess = info.activityInfo.processName;
            r.curComponent = component;
            if (r.callingUid != Process.SYSTEM_UID && isSingleton) {
                info.activityInfo =
                  mService.getActivityInfoForUser(info.activityInfo, 0);
            }
            r.curReceiver = info.activityInfo;
            if (DEBUG_MU && r.callingUid > UserHandle.PER_USER_RANGE) {
                Slog.v(TAG_MU,
                  "Updated broadcast record activity info for secondary user, "
                  + info.activityInfo + ", callingUid = " + r.callingUid + ", uid = "
                  + info.activityInfo.applicationInfo.uid);
            }
            try {
                AppGlobals.getPackageManager().setPackageStoppedState(
                  r.curComponent.getPackageName(), false,
                  UserHandle.getUserId (r.callingUid));
            } catch (RemoteException e) {
            } catch (IllegalArgumentException e) {
                Slog.w(TAG, "Failed trying to unstop package "
```

```
                + r.curComponent.getPackageName() + ": " + e);
        }
        ProcessRecord app = mService.getProcessRecordLocked(targetProcess,
          info.activityInfo.applicationInfo.uid);
        if (app!=null && app.thread!=null) {
            try {
                app.addPackage(info.activityInfo.packageName);
                processCurBroadcastLocked(r, app);
                return;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when sending broadcast to "
                  + r.curComponent, e);
            } catch (RuntimeException e) {
                Log.wtf(TAG, "Failed sending broadcast to "
                  + r.curComponent + " with " + r.intent, e);
                logBroadcastReceiverDiscardLocked(r);
                finishReceiverLocked(r, r.resultCode, r.resultData,
                  r.resultExtras, r.resultAbort, true);
                scheduleBroadcastsLocked();
                r.state = BroadcastRecord.IDLE;
                return;
            }
        }
        if (DEBUG_BROADCAST)
            Slog.v(TAG, "Need to start app ["
              + mQueueName + "] " + targetProcess + " for broadcast " + r);
        if ((r.curApp=mService.startProcessLocked(targetProcess,
          info.activityInfo.applicationInfo, true,
          r.intent.getFlags() | Intent.FLAG_FROM_BACKGROUND,
          "broadcast", r.curComponent,
          (r.intent.getFlags()&Intent.FLAG_RECEIVER_BOOT_UPGRADE) != 0,
          false)) == null) {
            Slog.w(TAG, "Unable to launch app "
              + info.activityInfo.applicationInfo.packageName + "/"
              + info.activityInfo.applicationInfo.uid + " for broadcast "
              + r.intent + ": process is bad");
            logBroadcastReceiverDiscardLocked(r);
            finishReceiverLocked(r, r.resultCode, r.resultData,
            r.resultExtras, r.resultAbort, true);
            scheduleBroadcastsLocked();
            r.state = BroadcastRecord.IDLE;
            return;
        }
        mPendingBroadcast = r;
        mPendingBroadcastRecvIndex = recIdx;
    }
}
```

函数 processNextBroadcast 的具体实现流程如下所示。

(1) 判断 fromMsg，如果是通过消息发送过来的，就为真，否则为假；如果为真，则 mBroadcastsScheduled = false，这样的话，在函数 scheduleBroadcastsLocked 里面就可以再次发送 BROADCAST_INTENT_MSG 的消息，从而触发 processNextBroadcast 函数被再次调用。

(2) 判断 mParallelBroadcasts 是否为空，不为空就开始调用这个列表里面的 receivers 来接收消息，这个过程后面在串行 Intent 的时候也会遇到，我们留到后面讨论，这里只需要知道它通过一个 while 循环把 Intent 发送给关注这个 Intent 的所有的 receivers。

(3) 判断 mPendingBroadcast 是否为空，如果不为空，就表示先前发送的串行的 Intent 还没有处理完毕，一般出现这种情况可能是因为我们要发送到的 receiver 还没有启动，所以需要先启动这个 Activity，然后等待这个 Activity 处理，这时候，这个 mPendingBroadcast 就为 true；如果是这种情况，需要判断这个 Activity 是否死了，如果死了，那么就把 mPendingBroadcast 设为 false，否则就直接返回，继续等待。

(4) 顺序地从 mOrderedBroadcasts 中取出 BroadCastRecord 消息，然后对这个消息的 receiver 一个一个地调用其接收流程。在处理这个消息的过程中，先判断其接收者是不是 BroadFilter，如果是，则调用 deliverToRegisteredReceiver 来接收。

(5) 如果不是 BroadCast Filter，则需要找出这个 receiver 所在的进程，这时通常是一个 IntentFilter 所在的进程。如果这个进程活着，则调用 processCurBroadcastLocked(r, app)来处理，否则需要用 startProcessLocked 先启动这个进程，然后设置 mPendingBroadcast = r，这样等应用启动后，它会处理这个消息。

10.2.7 检查权限

在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中，通过函数 deliverToRegisteredReceiverLocked 来检查广播发送和接收的权限，然后调用函数 performReceiveLocked 来进一步执行广播发送的操作。函数 deliverToRegisteredReceiverLocked 的具体实现代码如下所示：

```
private final void deliverToRegisteredReceiverLocked(BroadcastRecord r,
  BroadcastFilter filter, boolean ordered) {
    boolean skip = false;
    if (filter.requiredPermission != null) {
        int perm = mService.checkComponentPermission(filter.requiredPermission,
          r.callingPid, r.callingUid, -1, true);
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: broadcasting "
              + r.intent.toString()
              + " from " + r.callerPackage + " (pid="
              + r.callingPid + ", uid=" + r.callingUid + ")"
              + " requires " + filter.requiredPermission
              + " due to registered receiver " + filter);
            skip = true;
        }
    }
    if (!skip && r.requiredPermission!=null) {
        int perm = mService.checkComponentPermission(r.requiredPermission,
          filter.receiverList.pid, filter.receiverList.uid, -1, true);
```

```
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: receiving "
              + r.intent.toString()
              + " to " + filter.receiverList.app
              + " (pid=" + filter.receiverList.pid
              + ", uid=" + filter.receiverList.uid + ")"
              + " requires " + r.requiredPermission
              + " due to sender " + r.callerPackage
              + " (uid " + r.callingUid + ")");
            skip = true;
        }
    }
    if (r.appOp != AppOpsManager.OP_NONE) {
        int mode = mService.mAppOpsService.checkOperation(r.appOp,
          filter.receiverList.uid, filter.packageName);
        if (mode != AppOpsManager.MODE_ALLOWED) {
            if (DEBUG_BROADCAST)
                Slog.v(TAG,
                  "App op " + r.appOp + " not allowed for broadcast to uid "
                  + filter.receiverList.uid + " pkg " + filter.packageName);
            skip = true;
        }
    }

    if (!skip) {
        if (ordered) {
            r.receiver = filter.receiverList.receiver.asBinder();
            r.curFilter = filter;
            filter.receiverList.curBroadcast = r;
            r.state = BroadcastRecord.CALL_IN_RECEIVE;
            if (filter.receiverList.app != null) {
                r.curApp = filter.receiverList.app;
                filter.receiverList.app.curReceiver = r;
                mService.updateOomAdjLocked();
            }
        }
        try {
            if (DEBUG_BROADCAST_LIGHT) {
                int seq = r.intent.getIntExtra("seq", -1);
                Slog.i(TAG, "Delivering to " + filter
                  + " (seq=" + seq + "): " + r);
            }
            performReceiveLocked(filter.receiverList.app,
              filter.receiverList.receiver,
              new Intent(r.intent), r.resultCode, r.resultData,
              r.resultExtras, r.ordered, r.initialSticky, r.userId);
            if (ordered) {
                r.state = BroadcastRecord.CALL_DONE_RECEIVE;
            }
```

```
        } catch (RemoteException e) {
            Slog.w(TAG, "Failure sending broadcast " + r.intent, e);
            if (ordered) {
                r.receiver = null;
                r.curFilter = null;
                filter.receiverList.curBroadcast = null;
                if (filter.receiverList.app != null) {
                    filter.receiverList.app.curReceiver = null;
                }
            }
        }
    }
}
```

再看函数 performReceiveLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，具体实现代码如下所示：

```
private static void performReceiveLocked(ProcessRecord app,
  IIntentReceiver receiver, Intent intent, int resultCode, String data,
  Bundle extras, boolean ordered, boolean sticky, int sendingUser)
  throws RemoteException {
    // Send the intent to the receiver asynchronously using one-way binder calls.
    if (app!=null && app.thread!=null) {
        app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
          data, extras, ordered, sticky, sendingUser);
    } else {
        receiver.performReceive(intent, resultCode, data, extras, ordered,
          sticky, sendingUser);
    }
}
```

各个参数的具体说明如下所示。

- app：表示注册广播接收器的 Activity 所在的进程记录块。
- receiver：指向一个实现了 IIntentReceiver 接口的 Binder 代理对象，用来表示目标广播接收者。
- intent：表示即将要发送给目标广播接收者的一个广播。

10.2.8 处理的进程通信请求

先看函数 scheduleRegisteredReceiver，功能是通过 Binder 驱动程序来到类 ApplicationThread 的函数 scheduleRegisteredReceiver 中。

函数 scheduleRegisteredReceiver 在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，它的功能是把这个广播分发给 MainActivity，具体实现代码如下所示：

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
  int resultCode, String dataStr, Bundle extras, boolean ordered,
  boolean sticky, int sendingUser) throws RemoteException {
```

```
    Parcel data = Parcel.obtain();
    data.writeInterfaceToken(IApplicationThread.descriptor);
    data.writeStrongBinder(receiver.asBinder());
    intent.writeToParcel(data, 0);
    data.writeInt(resultCode);
    data.writeString(dataStr);
    data.writeBundle(extras);
    data.writeInt(ordered? 1 : 0);
    data.writeInt(sticky? 1 : 0);
    data.writeInt(sendingUser);
    mRemote.transact(SCHEDULE_REGISTERED_RECEIVER_TRANSACTION, data, null,
      IBinder.FLAG_ONEWAY);
    data.recycle();
}
```

再看文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中的函数 scheduleRegisteredReceiver，功能是通过 Binder 驱动程序进入到类 ApplicationThread 的函数 scheduleRegisteredReceiver，ApplicationThread 是 ActivityThread 的一个内部类。

函数 scheduleRegisteredReceiver 的具体实现代码如下所示：

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
  int resultCode, String dataStr, Bundle extras, boolean ordered,
  boolean sticky, int sendingUser) throws RemoteException {
    receiver.performReceive(intent, resultCode, dataStr, extras, ordered,
      sticky, sendingUser);
}
```

再看函数 performReceive，功能是接收来自参数 intent 描述的广播。函数 performReceive 在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void performReceive(Intent intent, int resultCode, String data,
  Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
    LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
    if (ActivityThread.DEBUG_BROADCAST) {
        int seq = intent.getIntExtra("seq", -1);
        Slog.i(ActivityThread.TAG, "Receiving broadcast " + intent.getAction()
          + " seq=" + seq + " to " + (rd!=null? rd.mReceiver : null));
    }
    if (rd != null) {
        rd.performReceive(intent, resultCode, data, extras,
          ordered, sticky, sendingUser);
    } else {
        if (ActivityThread.DEBUG_BROADCAST)
            Slog.i(ActivityThread.TAG,
              "Finishing broadcast to unregistered receiver");
        IActivityManager mgr = ActivityManagerNative.getDefault();
        try {
            if (extras != null) {
                extras.setAllowFds(false);
```

```
            }
            mgr.finishReceiver(this, resultCode, data, extras, false);
        } catch (RemoteException e) {
            Slog.w(ActivityThread.TAG,
              "Couldn't finish broadcast to unregistered receiver");
        }
    }
}
```

在上述代码中，调用类 ReceiverDispatcher 中的函数 performReceive 实现了进一步处理。下面看类 ReceiverDispatcher 中的函数 performReceive，此函数也是在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void performReceive(Intent intent, int resultCode, String data,
  Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
    if (ActivityThread.DEBUG_BROADCAST) {
        int seq = intent.getIntExtra("seq", -1);
        Slog.i(ActivityThread.TAG, "Enqueueing broadcast " + intent.getAction()
          + " seq=" + seq + " to " + mReceiver);
    }
    Args args = new Args(
      intent, resultCode, data, extras, ordered, sticky, sendingUser);
    if (!mActivityThread.post(args)) {
        if (mRegistered && ordered) {
            IActivityManager mgr = ActivityManagerNative.getDefault();
            if (ActivityThread.DEBUG_BROADCAST)
                Slog.i(ActivityThread.TAG,
                  "Finishing sync broadcast to " + mReceiver);
            args.sendFinished(mgr);
        }
    }
}
```

这样，ReceiverDispatcher 的内部类 Args 会在 MainActivity 所在的线程消息循环中处理这个广播，并最终将这个广播分发给所注册的 BroadcastReceiver 实例的函数 onReceive 进行处理。

接下来，我们看类 Args 中的函数 run，此函数在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void run() {
    final BroadcastReceiver receiver = mReceiver;
    final boolean ordered = mOrdered;
    if (ActivityThread.DEBUG_BROADCAST) {
        int seq = mCurIntent.getIntExtra("seq", -1);
        Slog.i(ActivityThread.TAG, "Dispatching broadcast "
          + mCurIntent.getAction() + " seq=" + seq + " to " + mReceiver);
        Slog.i(ActivityThread.TAG,
          " mRegistered=" + mRegistered + " mOrderedHint=" + ordered);
    }
    final IActivityManager mgr = ActivityManagerNative.getDefault();
```

```
    final Intent intent = mCurIntent;
    mCurIntent = null;

    if (receiver == null || mForgotten) {
        if (mRegistered && ordered) {
            if (ActivityThread.DEBUG_BROADCAST)
                Slog.i(ActivityThread.TAG,
                  "Finishing null broadcast to " + mReceiver);
            sendFinished(mgr);
        }
        return;
    }
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "broadcastReceiveReg");
    try {
        ClassLoader cl =  mReceiver.getClass().getClassLoader();
        intent.setExtrasClassLoader(cl);
        setExtrasClassLoader(cl);
        receiver.setPendingResult(this);
        receiver.onReceive(mContext, intent);
    } catch (Exception e) {
        if (mRegistered && ordered) {
            if (ActivityThread.DEBUG_BROADCAST)
                Slog.i(ActivityThread.TAG,
                  "Finishing failed broadcast to " + mReceiver);
            sendFinished(mgr);
        }
        if (mInstrumentation == null
          || !mInstrumentation.onException(mReceiver, e)) {
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            throw new RuntimeException(
              "Error receiving broadcast " + intent
              + " in " + mReceiver, e);
        }
    }
    if (receiver.getPendingResult() != null) {
        finish();
    }
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
}
```

上述代码中，mReceiver 是类 ReceiverDispatcher 的成员变量，类型是 BroadcastReceiver。此时，通过这个 ReceiverDispatcher 实例，即可调用它的 onReceive 函数分发处理这个广播。

10.3　分析BroadcastReceiver

在 Android 的广播系统中，ActivityManagerService 是整个广播的中心，负责系统中所有广播的注册和发布操作。其中 Android 应用程序注册广播接收器的过程，就是将广播接收器注册

到 ActivityManagerService 的过程。在 Android 应用程序中，通过调用类 ContextWrapper 中的函数 registerReceiver 将广播接收器 BroadcastReceiver 注册到 ActivityManagerService 中，而类 ContextWrapper 本身可以使用类 ContextImpl 来注册广播接收器。

10.3.1 MainActivity的调用

在 Android 的广播系统中，从 MainActivity 开始发起注册广播接收器的操作。

类 MainActivity 中，调用函数 registerReceiver 来注册广播接收器，对应的代码如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ...
    public void onResume() {
        super.onResume();
        IntentFilter counterActionFilter =
          new IntentFilter(CounterService.BROADCAST_COUNTER_ACTION);
        registerReceiver(counterActionReceiver, counterActionFilter);
    }
```

在上述代码中，函数 onResume 通过其父类 ContextWrapper 中的函数 registerReceiver 注册一个 BroadcastReceiver 实例 counterActionReceiver，并通过 IntentFilter 实例 counterActionFilter 通知 ActivityManagerService 要订阅的广播是 CounterService.BROADCAST_COUNTER_ACTION 类型。这样,当 ActivityManagerService 收到 CounterService.BROADCAST_COUNTER_ACTION 类型的广播时，就会分发给 counterActionReceiver 实例的 onReceive 函数。

10.3.2 注册广播接收者

再看类 ContextWrapper 中的函数 registerReceiver，功能是在 ActivityManagerService 中注册广播接收者。

函数 registerReceiver 在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义，具体实现代码如下所示：

```
public Intent registerReceiver(BroadcastReceiver receiver,
  IntentFilter filter) {
    return mBase.registerReceiver(receiver, filter);
}
```

在上述代码中，其实是调用了类 ContextImpl 中的函数 registerReceiver 来注册广播接收者。函数 registerReceiver 在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义，具体实现代码如下所示：

```
public Intent registerReceiver(BroadcastReceiver receiver,
  IntentFilter filter) {
    return registerReceiver(receiver, filter, null, null);
}

@Override
public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,
```

```
  String broadcastPermission, Handler scheduler) {
    if (receiver == null) {
        return super.registerReceiver(
          null, filter, broadcastPermission, scheduler);
    } else {
        throw new ReceiverCallNotAllowedException(
          "BroadcastReceiver components are not allowed "
          + "to register to receive intents");
    }
}
private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
  IntentFilter filter, String broadcastPermission,
  Handler scheduler, Context context) {
    IIntentReceiver rd = null;
    if (receiver != null) {
        if (mPackageInfo!=null && context!=null) {
            if (scheduler == null) {
                scheduler = mMainThread.getHandler();
            }
            rd = mPackageInfo.getReceiverDispatcher(
              receiver, context, scheduler,
              mMainThread.getInstrumentation(), true);
        } else {
            if (scheduler == null) {
                scheduler = mMainThread.getHandler();
            }
            rd = new LoadedApk.ReceiverDispatcher(
              receiver, context, scheduler, null, true).getIIntentReceiver();
        }
    }
    try {
        return ActivityManagerNative.getDefault().registerReceiver(
          mMainThread.getApplicationThread(), mBasePackageName,
          rd, filter, broadcastPermission, userId);
    } catch (RemoteException e) {
        return null;
    }
}
```

在上述代码中，变量 mPackageInfo 是一个 LoadedApk 实例，功能是处理广播接收者的信息。参数 broadcastPermission 和 scheduler 都是 null，而参数 context 是通过调用函数 getOuterContext 得到的，在此处指向了 MainActivity。

10.3.3 获取接口对象

函数 getReceiverDispatcher 的功能是获得一个 IIntentReceiver 接口对象 rd，这是一个 Binder 对象。然后将这个对象传给 ActivityManagerService，ActivityManagerService 在收到相应的广播时，会通过这个 Binder 对象通知 MainActivity 来接收。函数 getReceiverDispatcher 在文件

frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
  Context context, Handler handler,
  Instrumentation instrumentation, boolean registered) {
    synchronized(mReceivers) {
        LoadedApk.ReceiverDispatcher rd = null;
        HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = null;
        if (registered) {
            map = mReceivers.get(context);
            if (map != null) {
                rd = map.get(r);
            }
        }
        if (rd == null) {
            rd = new ReceiverDispatcher(r, context, handler,
              instrumentation, registered);
            if (registered) {
                if (map == null) {
                    map = new HashMap<BroadcastReceiver,
                      LoadedApk.ReceiverDispatcher>();
                    mReceivers.put(context, map);
                }
                map.put(r, rd);
            }
        } else {
            rd.validate(context, handler);
        }
        rd.mForgotten = false;
        return rd.getIIntentReceiver();
    }
}
static final class ReceiverDispatcher {
    final static class InnerReceiver extends IIntentReceiver.Stub {
        final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
        final LoadedApk.ReceiverDispatcher mStrongRef;

        InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
            mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
            mStrongRef=strong? rd : null;
        }
        public void performReceive(Intent intent, int resultCode, String data,
          Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
            LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
            if (ActivityThread.DEBUG_BROADCAST) {
                int seq = intent.getIntExtra("seq", -1);
                Slog.i(ActivityThread.TAG, "Receiving broadcast "
                  + intent.getAction() + " seq=" + seq
                  + " to " + (rd != null ? rd.mReceiver : null));
```

```
            }
            if (rd != null) {
                rd.performReceive(intent, resultCode, data, extras,
                  ordered, sticky, sendingUser);
            } else {
                if (ActivityThread.DEBUG_BROADCAST)
                    Slog.i(ActivityThread.TAG,
                      "Finishing broadcast to unregistered receiver");
                IActivityManager mgr = ActivityManagerNative.getDefault();
                try {
                    if (extras != null) {
                        extras.setAllowFds(false);
                    }
                    mgr.finishReceiver(this, resultCode, data, extras, false);
                } catch (RemoteException e) {
                    Slog.w(ActivityThread.TAG,
                      "Couldn't finish broadcast to unregistered receiver");
                }
            }
        }
    }
    final IIntentReceiver.Stub mIIntentReceiver;
    final BroadcastReceiver mReceiver;
    final Context mContext;
    final Handler mActivityThread;
    final Instrumentation mInstrumentation;
    final boolean mRegistered;
    final IntentReceiverLeaked mLocation;
    RuntimeException mUnregisterLocation;
    boolean mForgotten;
    ...
    ReceiverDispatcher(BroadcastReceiver receiver, Context context,
      Handler activityThread, Instrumentation instrumentation,
      boolean registered) {
        if (activityThread == null) {
            throw new NullPointerException("Handler must not be null");
        }

        mIIntentReceiver = new InnerReceiver(this, !registered);
        mReceiver = receiver;
        mContext = context;
        mActivityThread = activityThread;
        mInstrumentation = instrumentation;
        mRegistered = registered;
        mLocation = new IntentReceiverLeaked(null);
        mLocation.fillInStackTrace();
    }
}
```

在上述函数 getReceiverDispatcher 中，先检查参数 r 是否已经有相应的 ReceiverDispatcher，如果有，就直接返回，否则就新建一个 ReceiverDispatcher，并且以 r 为 Key 值保存在一个 HashMap 中。只要给定一个 Activity 和 BroadcastReceiver，就可以查看在 LoadedApk 中是否已经存在相应的广播接收发布器 ReceiverDispatcher。

接下来，看文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中的函数 registerReceiver，具体实现代码如下所示：

```
public Intent registerReceiver(IApplicationThread caller, String packageName,
  IIntentReceiver receiver, IntentFilter filter, String perm, int userId)
  throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller!=null? caller.asBinder() : null);
    data.writeString(packageName);
    data.writeStrongBinder(receiver!=null? receiver.asBinder() : null);
    filter.writeToParcel(data, 0);
    data.writeString(perm);
    data.writeInt(userId);
    mRemote.transact(REGISTER_RECEIVER_TRANSACTION, data, reply, 0);
    reply.readException();
    Intent intent = null;
    int haveIntent = reply.readInt();
    if (haveIntent != 0) {
        intent = Intent.CREATOR.createFromParcel(reply);
    }
    reply.recycle();
    data.recycle();
    return intent;
}
```

上述代码通过 Binder 驱动程序来到 ActivityManagerService 的 registerReceiver 函数中。

10.3.4 处理进程间的通信请求

接下来看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 registerReceiver，功能是处理组件 Broadcounter 发出的类型为 REGISTER_RECEIVER_TRANSACTION 的进程间的通信请求。函数 registerReceiver 的具体实现代码如下所示：

```
public Intent registerReceiver(IApplicationThread caller, String callerPackage,
  IIntentReceiver receiver, IntentFilter filter,
  String permission, int userId) {
    enforceNotIsolatedCaller("registerReceiver");
    int callingUid;
    int callingPid;
    synchronized(this) {
        //调用函数 registerReceiver 的应用程序进程记录块，MainActivity 就是在里面启动的
        ProcessRecord callerApp = null;
```

```
        if (caller != null) {
            callerApp = getRecordForAppLocked(caller);
            if (callerApp == null) {
                throw new SecurityException(
                  "Unable to find app for caller " + caller
                  + " (pid=" + Binder.getCallingPid()
                  + ") when registering receiver " + receiver);
            }
            if (callerApp.info.uid != Process.SYSTEM_UID
              && !callerApp.pkgList.contains(callerPackage)) {
                throw new SecurityException("Given caller package " + callerPackage
                  + " is not running in process " + callerApp);
            }
            callingUid = callerApp.info.uid;
            callingPid = callerApp.pid;
        } else {
            callerPackage = null;
            callingUid = Binder.getCallingUid();
            callingPid = Binder.getCallingPid();
        }

        userId = this.handleIncomingUser(callingPid, callingUid, userId,
          true, true, "registerReceiver", callerPackage);
        //传进来的 filter 只有一个 action，就是前面描述的
        //CounterService.BROADCAST_COUNTER_ACTION 了
        List allSticky = null;

        // Look for any matching sticky broadcasts...
        Iterator actions = filter.actionsIterator();
        if (actions != null) {
            while (actions.hasNext()) {
                String action = (String)actions.next();
                //通过函数 getStickiesLocked 查找是否存在对应的 sticky intent 列表
                allSticky = getStickiesLocked(
                  action, filter, allSticky, UserHandle.USER_ALL);
                allSticky = getStickiesLocked(
                  action, filter, allSticky, UserHandle.getUserId(callingUid));
            }
        } else {
            allSticky = getStickiesLocked(
              null, filter, allSticky, UserHandle.USER_ALL);
            allSticky = getStickiesLocked(
              null, filter, allSticky, UserHandle.getUserId(callingUid));
        }

        Intent sticky = allSticky!=null? (Intent)allSticky.get(0) : null;

        if (DEBUG_BROADCAST)
            Slog.v(TAG, "Register receiver " + filter + ": " + sticky);
```

```
if (receiver == null) {
    return sticky;
}
//如果传进来的 receiver 不为 null，则继续往下执行
ReceiverList rl =
  (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
if (rl == null) {
    rl = new ReceiverList(
      this, callerApp, callingPid, callingUid, userId, receiver);
    if (rl.app != null) {
        rl.app.receivers.add(rl);
    } else {
        try {
            receiver.asBinder().linkToDeath(rl, 0);
        } catch (RemoteException e) {
            return sticky;
        }
        rl.linkedToDeath = true;
    }
    mRegisteredReceivers.put(receiver.asBinder(), rl);
} else if (rl.uid != callingUid) {
    throw new IllegalArgumentException(
      "Receiver requested to register for uid " + callingUid
      + " was previously registered for uid " + rl.uid);
} else if (rl.pid != callingPid) {
    throw new IllegalArgumentException(
      "Receiver requested to register for pid " + callingPid
      + " was previously registered for pid " + rl.pid);
} else if (rl.userId != userId) {
    throw new IllegalArgumentException(
      "Receiver requested to register for user " + userId
      + " was previously registered for user " + rl.userId);
}
//将广播接收器 receiver 保存起来，并没有把它与 filter 关联起来
BroadcastFilter bf = new BroadcastFilter(
  filter, rl, callerPackage, permission, callingUid, userId);
rl.add(bf);
if (!bf.debugCheck()) {
    Slog.w(TAG, "==> For Dynamic broadast");
}
mReceiverResolver.addFilter(bf);

// Enqueue broadcasts for all existing stickies that match
// this filter.
if (allSticky != null) {
    ArrayList receivers = new ArrayList();
    receivers.add(bf);
```

```
            int N = allSticky.size();
            for (int i=0; i<N; i++) {
                Intent intent = (Intent)allSticky.get(i);
                BroadcastQueue queue = broadcastQueueForIntent(intent);
                BroadcastRecord r = new BroadcastRecord(queue, intent, null,
                  null, -1, -1, null, AppOpsManager.OP_NONE, receivers, null, 0,
                  null, null, false, true, true, -1);
                queue.enqueueParallelBroadcastLocked(r);
                queue.scheduleBroadcastsLocked();
            }
        }
        return sticky;
    }
}
```

在上述代码中，使用创建的 BroadcastFilter 把广播接收器列表 rl 和 filter 关联起来，然后保存在 ActivityManagerService 的成员变量 mReceiverResolver 中。

第 11 章
多媒体系统详解

从 Android 2.2 版本以后，Android 对多媒体框架进行了很大的调整，抛弃了原来的 OpenCore 框架，改用 Stagefright 框架，仅仅对 OpenCore 中的 omx-component 部分做了引用。与 OpenCore 框架相比，Stagefright 框架更加易懂，并且封装也相对简单。在 Android 2.2 版本及以前，OpenCore 位于 external 目录下，在 Android 2.3 以后，多媒体的功能被放置到 frameworks/base/media 目录下。

本章将详细讲解 Android 系统中 OpenCore 框架和 Stagefright 框架的基本知识，为读者步入本书后面知识的学习打下基础。

11.1 Android多媒体系统介绍

在 Android 的多媒体系统中，可以根据需要添加一些第三方插件，这样可以增强多媒体系统的功能。在 Android 系统的本地多媒体引擎上面，是 Android 的多媒体本地框架，而在多媒体本地框架上面是多媒体 JNI 和多媒体的 Java 框架部分。与多媒体相关的应用程序通过调用 Android Java 框架层，来提供标准的多媒体 API 进行构建。我们本章将要讲解的 OpenCore 引擎和 Stagefright 引擎是 Android 本地框架中定义接口的实现者，上层调用者不知道 Android 下层使用什么多媒体引擎。

Android 多媒体引擎和插件的基本层次结构如图 11-1 所示。

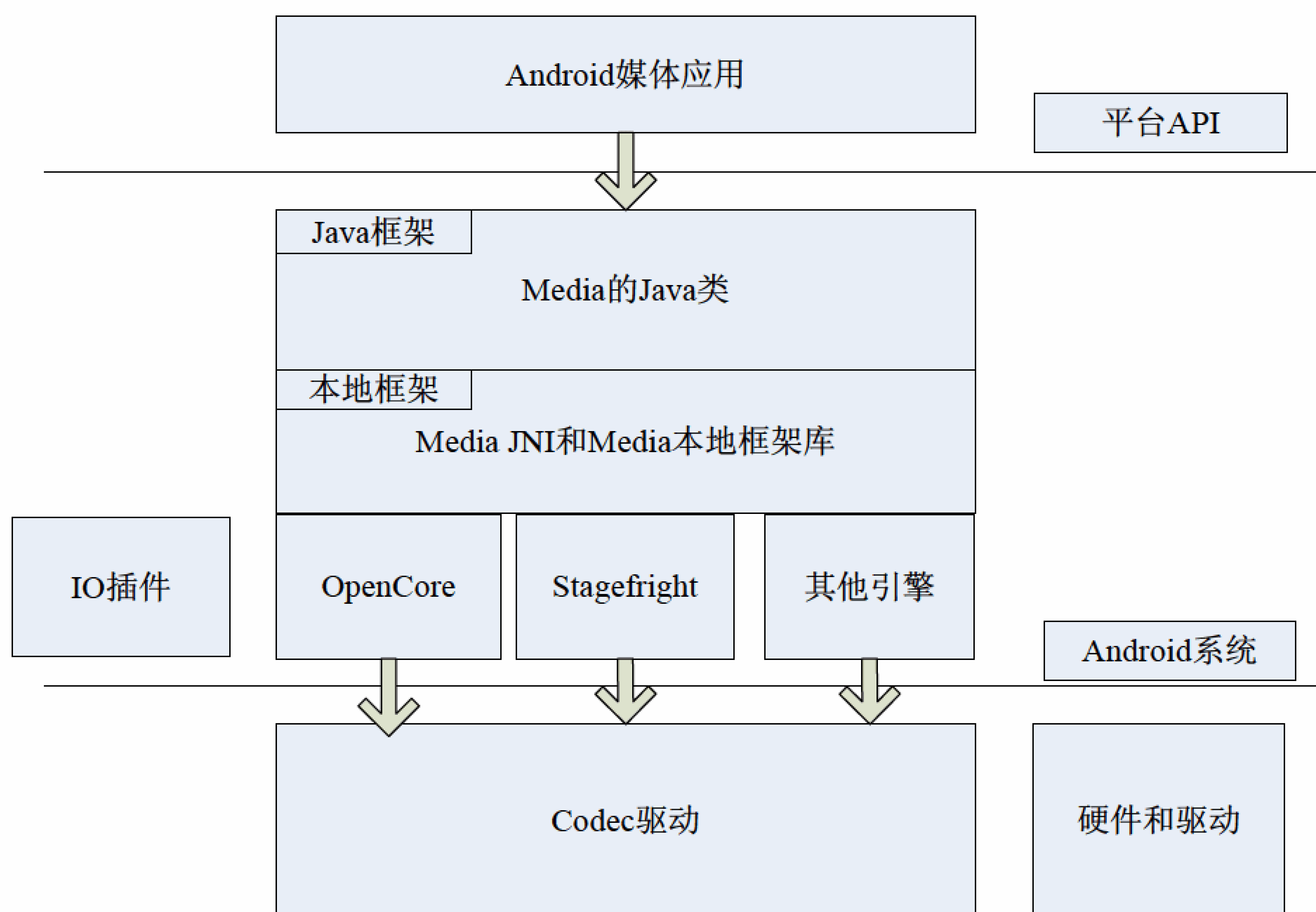


图 11-1　Android多媒体引擎和插件的基本层次结构

Android 系统的多媒体框架系统如图 11-2 所示。

从多媒体应用的实现角度来看，多媒体系统主要包含如下两方面的内容。

- 输入输出环节：音频、视频纯数据流的输入、输出系统。
- 中间处理环节：包括文件格式处理和编码/解码环节处理。

假如想要处理一个 MP3 文件，媒体播放器的处理流程是：将一个 MP3 格式的文件作为播放器的输入，将声音从播放器设备输出。在具体实现上，MP3 播放器经过了 MP3 格式文件解析、MP3 码流解码和 PCM 输出播放的过程，整个过程如图 11-3 所示。

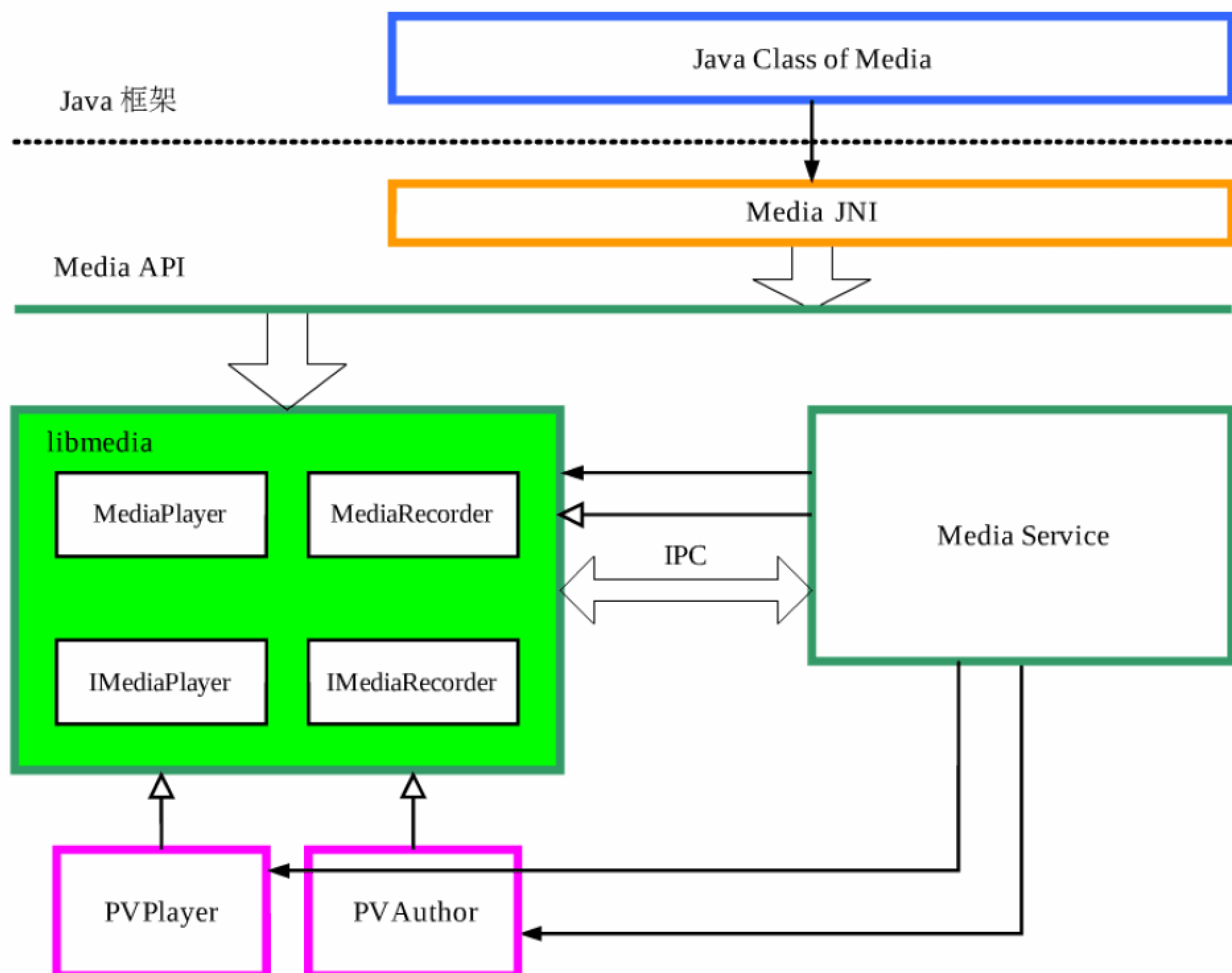


图 11-2　Android系统的多媒体框架结构

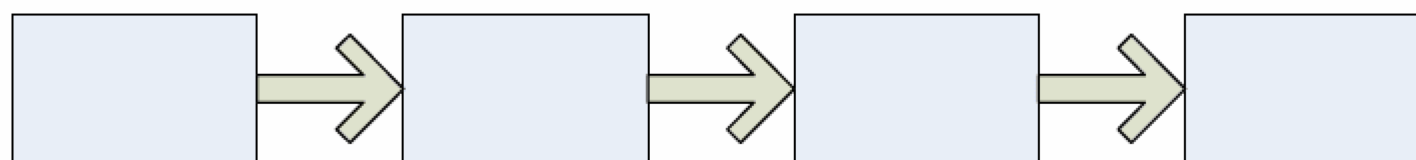


图 11-3　MP3 播放器的结构

11.2　OpenMAX框架详解

2006 年，NVIDIA 公司和 Khronos 组织联合推出了 OpenMAX，这是多媒体应用程序的框架标准。OpenMAX 是无授权费的、跨平台的应用程序接口 API。

OpenMAX 通过使媒体加速组件能够在开发、集成和编程环节中实现跨多操作系统和处理器硬件平台，提供全面的流媒体编码/解码器和应用程序便携化。

OpenMAX 的官方网站地址如下所示：

```
http://www.khronos.org/openmax/
```

OpenMAX 是一个多媒体应用程序的框架标准。在这个标准中，在集成层：OpenMAX IL 中定义了媒体组件接口，通过这些接口，可以在嵌入式器件的流媒体框架中，实现对加速编码器和解码器的快速集成。

Android 系统本身没有独立的多媒体系统，而是直接使用了市面中现成的产品，OpenMAX IL 便是其中之一。在 Android 结构中，OpenMAX IL 通常被当作多媒体引擎插件来使用。Android 最早的多媒体引擎是 OpenCore，后续版本逐渐使用 Stagefright 来代替。这两种引擎都可以使用 OpenMAX 作为插件，主要实现编码和解码(Codec)处理。

在 Android 的框架层中定义了由 Android 封装的 OpenMAX 接口，此接口与标准的接口类似。但是因为使用的是 C++类型接口，并且使用了 Android 的 Binder IPC 机制，所以处理速度会很快。后续引擎 Stagefright 使用了封装的 OpenMAX 接口，原先的引擎 OpenCore 并没有使用此接口，而是使用其他形式封装了 OpenMAX IL 层接口。

Android 中 OpenMAX 的基本层次结构如图 11-4 所示。

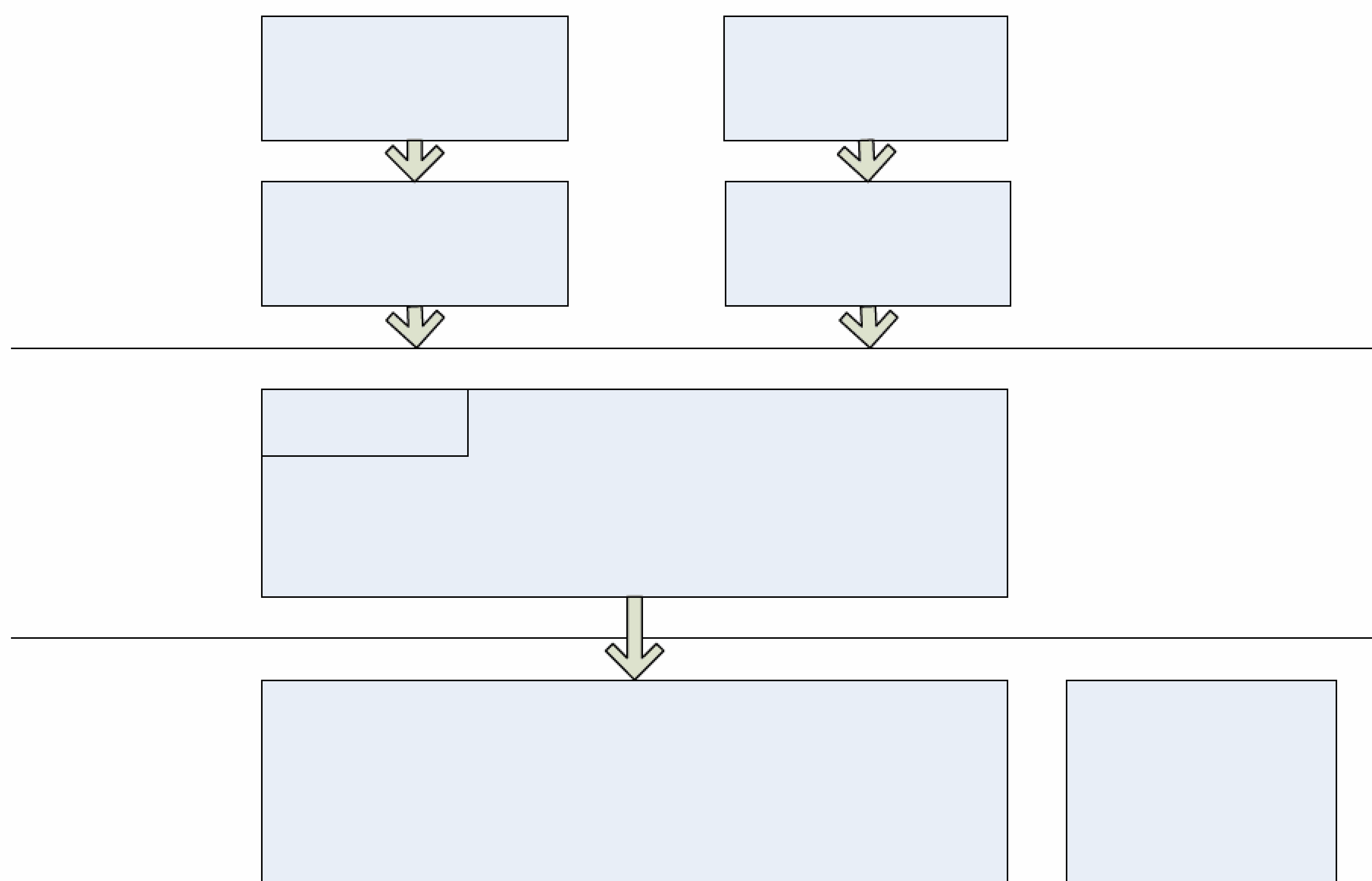


图 11-4　OpenMAX多媒体框架的层次结构

11.2.1　分析OpenMAX框架构成

图 11-4 中列出了 OpenMAX 多媒体框架的层次结构，在接下来的内容中，将详细讲解各个层次结构的基本知识。

1. OpenMAX总体层次结构

OpenMAX 分成三个层次，从上到下分别是 OpenMAX DL(开发层)、OpenMAX IL(集成层)和 OpenMAX AL(应用层)。在实际的应用中，OpenMAX 的三个层次中使用较多的是 OpenMAX IL 集成层，由于操作系统到硬件的差异和多媒体应用的差异，OpenMAX 的 DL 和 AL 层使用相对较少。

接下来给出上述三个层次结构的具体说明。

(1)　OpenMAX DL(Development Layer，开发层)

在 OpenMAX DL 中，定义了包含音频、视频和图像功能的 API，这样，供应商可以在一

个新的处理器上实现并优化，然后编写更广泛的编码/解码功能。OpenMAX DL 可以处理 FFT 和 Filter 等音频信号，也可以实现颜色空间转换和处理原始视频，并且可以实现对诸如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编码/解码的优化。

(2)　OpenMAX IL(Integration Layer，集成层)

OpenMAX IL 是一种音频、视频和图像编码/解码器，能够实现与多媒体编码/解码器的交互。OpenMAX IL 的主要目的是使用特征集合为编码/解码器提供一个系统抽象，以解决多个不同媒体系统之间的轻便性交互问题。

(3)　OpenMAX AL(Application Layer，应用层)

OpenMAX AL API 在应用程序和多媒体中间件之间提供了一个标准化接口，多媒体中间件提供服务以实现被期待的 API 功能。OpenMAX 具有三个层次，具体如图 11-5 所示。

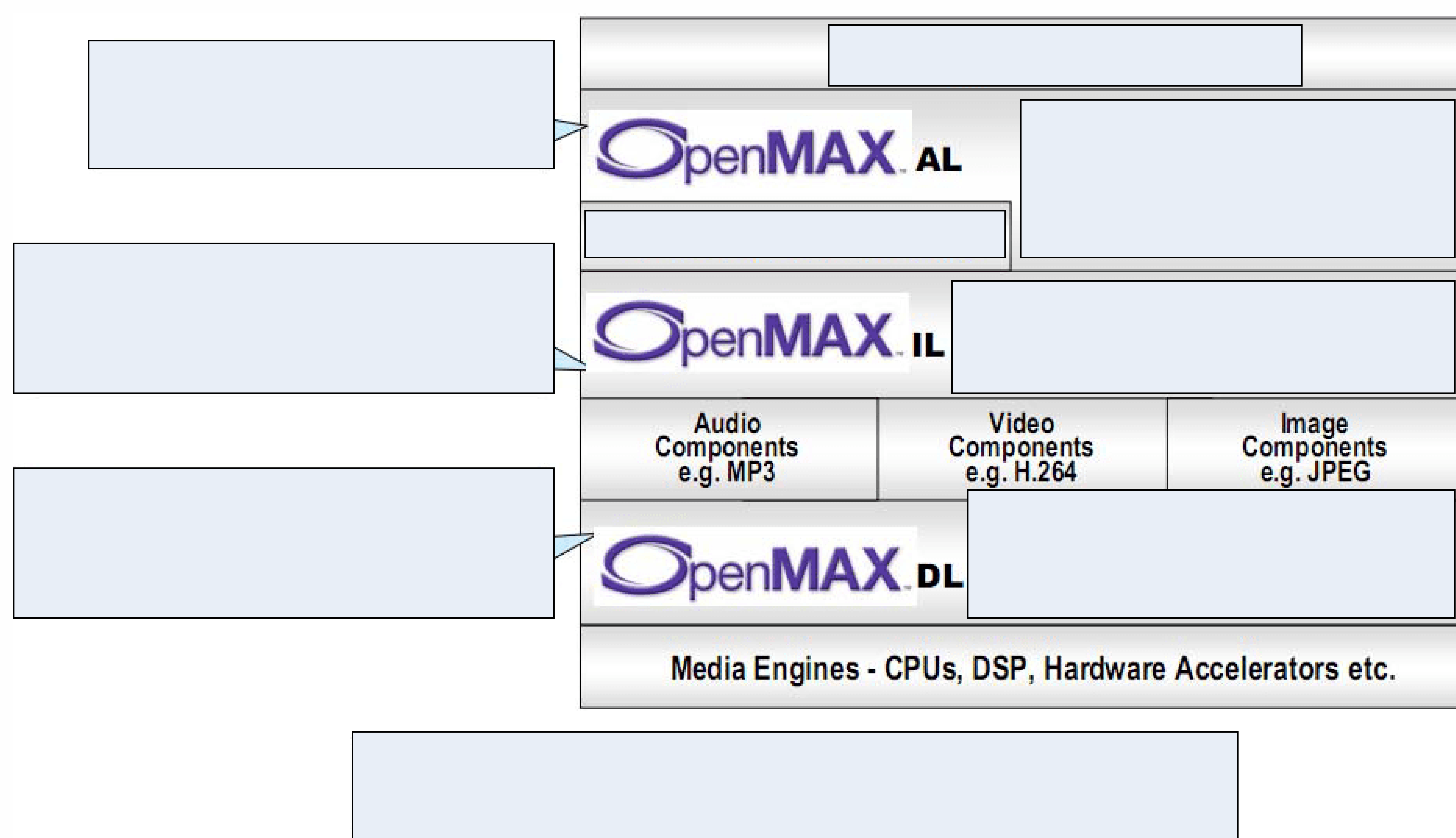


图 11-5　OpenMAX的层次

2. OpenMAX IL层的结构

在当前多媒体领域，因为 OpenMAX IL 的普及，其实际上已经成为多媒体框架标准。大多数嵌入式处理器或者多媒体编码/解码器模块的硬件生产者，通常都提供了标准的 OpenMAX IL 层的软件接口，这样程序员就可以基于此层次的标准化接口进行多媒体程序的开发。

OpenMAX IL 的接口层次结构比较科学，既不是硬件编码/解码器的接口，也不是应用程序层的接口，所以可以比较容易地实现标准化。OpenMAX IL 的层次结构如图 11-6 所示。

在图 11-6 表示的层次结构中，虚线部分里面的内容是 OpenMAX IL 层的内容，功能是实现 OpenMAX IL 中的各个组件(Component)。对于下层而言，OpenMAX IL 既可以调用 OpenMAX DL 层的接口，也可以直接调用各种 Codec 实现。对于上层而言，OpenMAX IL 既可以给 OpenMAX AL 层等框架层(Middleware)调用，也可以给应用程序直接调用。

OpenMAX IL 层中包含的主要内容如下所示。

- Client：客户端，OpenMAX IL 的调用者。
- Component：组件，OpenMAX IL 的单元，每一个组件实现一种功能。

- 端口(Port)：组件的输入输出接口。
- Tunneled：隧道化，让两个组件直接连接的方式。

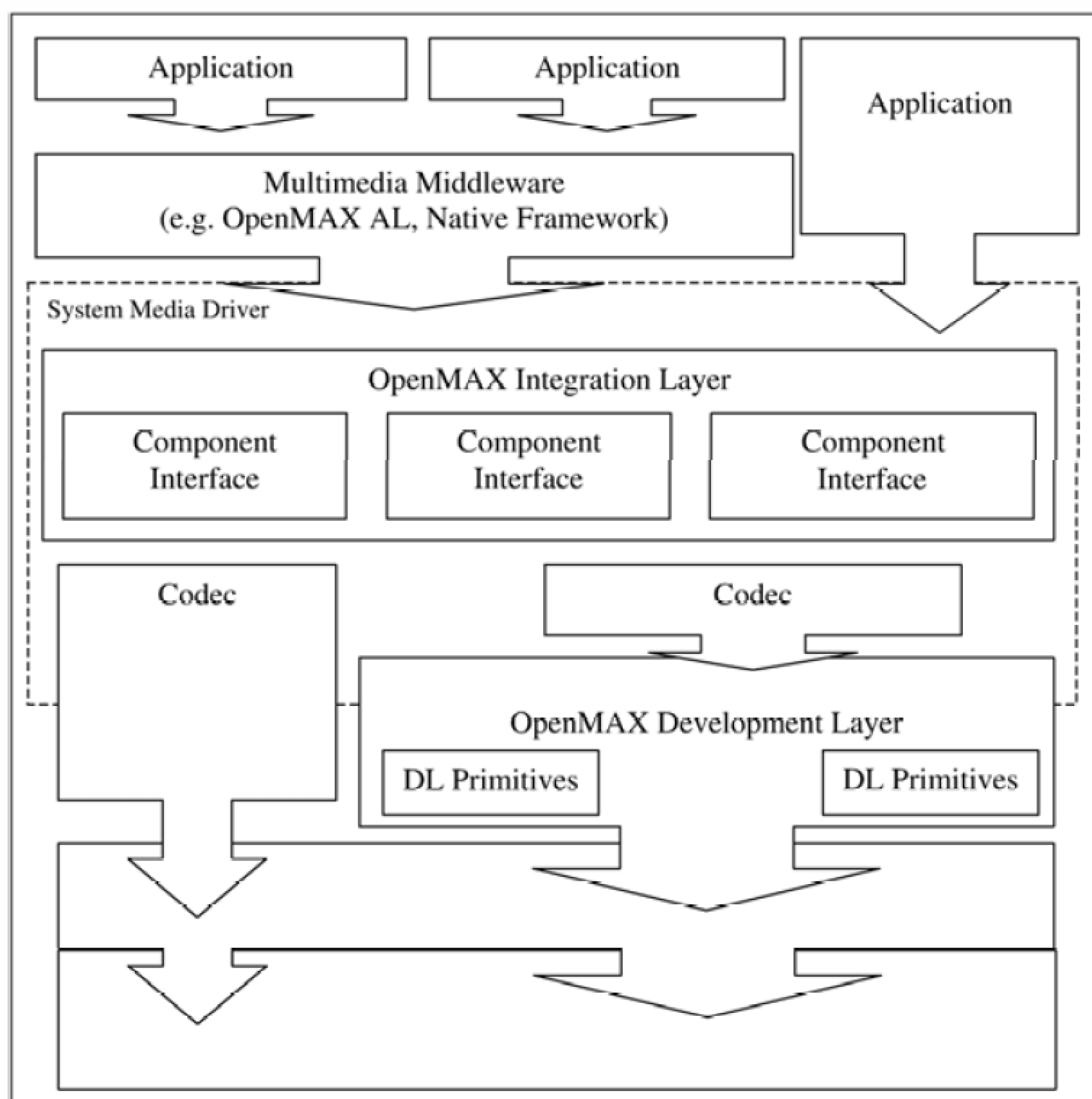


图 11-6　OpenMAX IL的层次结构

OpenMAX IL 层的运作流程如图 11-7 所示。

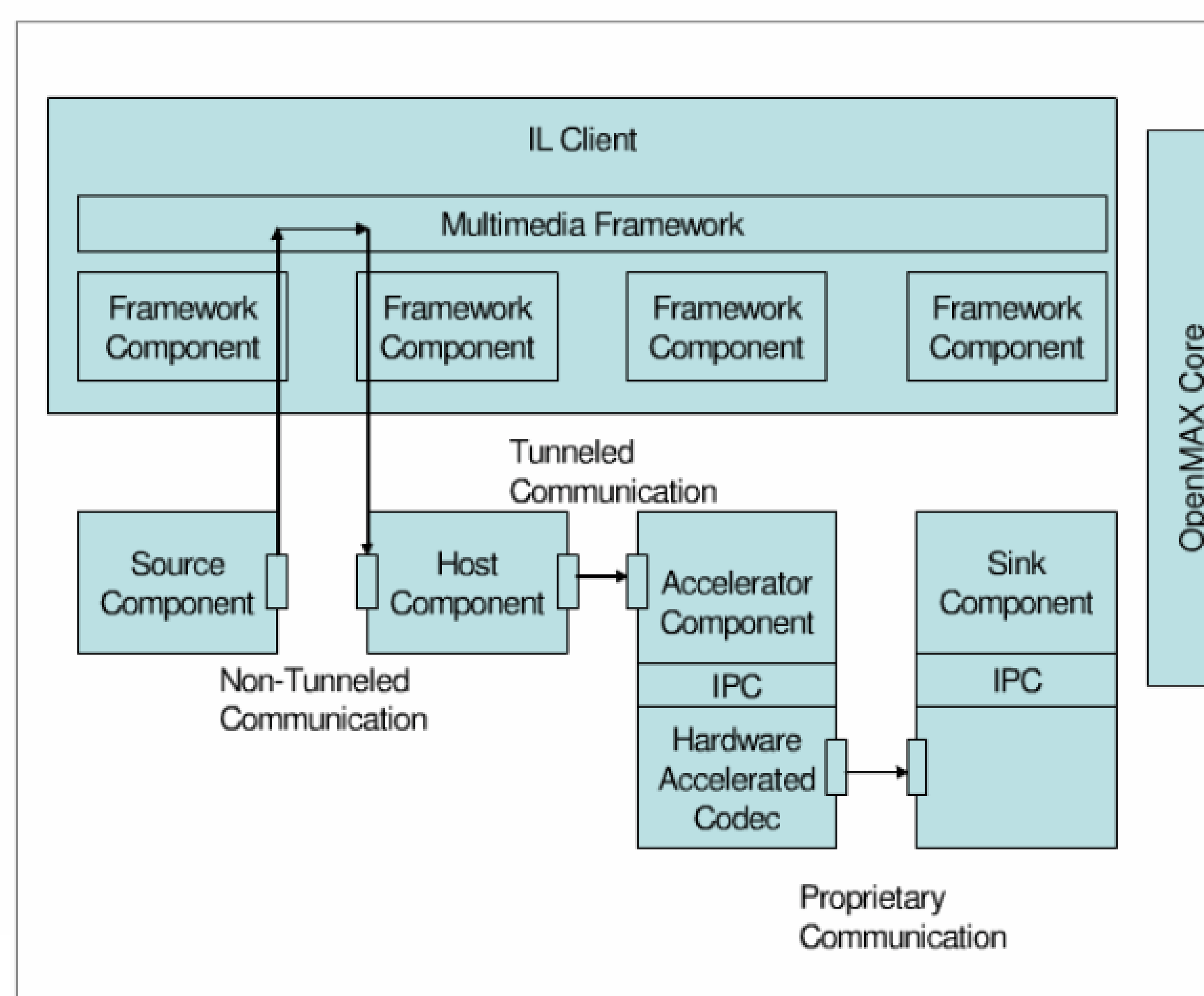


图 11-7　OpenMAX IL层的运作流程

在图 11-7 中，OpenMAX IL 层的客户端通过调用如下四个 OpenMAX IL 组件来实现同一个功能。

- Source 组件：只有一个输出端口。
- Host 组件：有一个输入端口和一个输出端口；
- Accelerator 组件：具有一个输入端口，调用了硬件的编码/解码器，加速主要体现在此环节上；
- Sink 组件：Accelerator 组件和 Sink 组件通过私有通信方式在内部进行连接，没有经过明确的组件端口。

在使用 OpenMAX IL 的时候，既可以经由客户端处理数据流，也可以不经由客户端处理数据流。在图 11-4 中，Source 组件到 Host 组件的数据流就是经过客户端的；而 Host 组件到 Accelerator 组件的数据流就没有经过客户端，使用了隧道化的方式；Accelerator 组件和 Sink 组件甚至可以使用私有的通信方式。

OpenMAX Core 是辅助组件正常运行的模块，它的任务是完成各个组件的初始化等工作。在具体运行时，需要重点初始化 OpenMAX IL 组件，而不是初始化 OpenMAX Core 组件。

在 OpenMAX IL 层中，真正的核心内容是 OpenMAX IL 组件，此组件分别以输入端和输出端为接口，端口可以被连接到另一个组件上。外部对组件可以发送命令，还可以设置/获取参数、进行配置等。组件的端口可以包含缓冲区(Buffer)的队列。

在 OpenMAX IL 层中，组件的处理的核心内容是通过输入端口来消耗 Buffer，通过输出端口来填充 Buffer，这样做的好处是通过多个组件的相互联接而构成流式处理。在 OpenMAX IL 层中，一个组件的基本结构如图 11-8 所示。

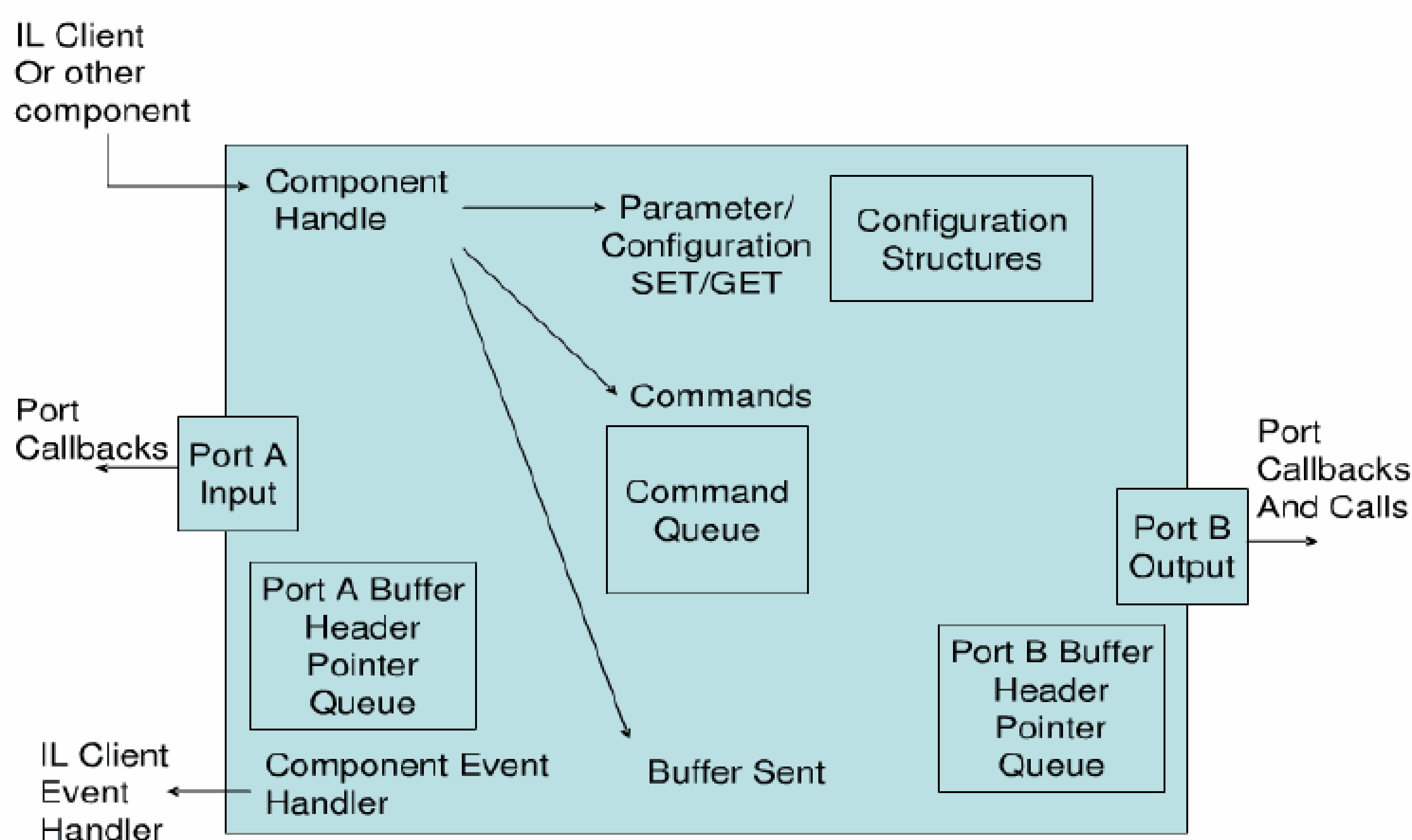


图 11-8　OpenMAX IL层中的组件结构

组件的功能与定义端口的类型有着密切的联系，在大多数情况下的具体联系如下所示：

- 只有一个输出端口的是 Source 组件。
- 只有一个输入端口的是 Sink 组件。
- 有多个输入端口、一个输出端口的是 Mux 组件。
- 有一个输入端口、多个输出端口的是 DeMux 组件。输入和输出端口各一个组件的为中间处理环节，这是最常见的组件。

端口根据应用来支持不同的数据类型。假如在输入端和输出端各有一个组件，如果输入端口使用的是MP3格式数据，而在输出端口使用的是PCM格式数据，那么此组件就是一个MP3解码组件。

注意：上述组件连接的方式有一个专业术语——隧道化，通过隧道化(Tunneled)的方式可以将不同的组件的一个输入端口和一个输出端口连接到一起，此时会合并两个组件的处理过程并实现共同处理，合二为一。

3. Android中的OpenMAX

在Android系统中，主要使用的是标准OpenMAX IL层的接口，在里面只是进行了简单的封装。通过使用标准的OpenMAX IL实现，可以很容易地将OpenMAX IL以插件的形式加入到Android系统中。

无论是OpenCore引擎，还是Stagefright引擎，都可以使用OpenMAX作为多媒体编码/解码器的插件，但是并没有直接使用OpenMAX IL层提供的纯C的接口，而只是对其进行了简单的封装处理。

Android系统对OpenMAX支持的力度逐渐扩大，在Android 2.x版本之后，Android的框架层开始封装定义OpenMAX IL层接口，甚至使用Android中的Binder IPC机制来调用。在Stagefright中使用了OpenMAX IL层接口，但是没有使用OpenCore。

OpenCore使用在OpenMAX IL层，作为编码/解码器插件，Android框架层封装OpenMAX接口的做法在后面的版本中才引入。

在Android系统中，主要使用了OpenMAX的编码/解码器功能。在Android系统中，使用的最多的仍然是编码/解码器组件，尽管OpenMAX也可以生成输入、输出、文件解析/构建等组件。主要原因有如下两条：

- 媒体输入/输出环节与系统有很大的关系，如果硬要使用OpenMAX标准，会比较麻烦。
- 文件解析/构建环节一般不需要使用硬件加速。因为编码/解码器组件最能体现硬件加速环节，所以最常使用。

在Android系统中，当实现OpenMAX IL层和标准的OpenMAX IL层时，需要实现如下两个环节。

(1) 编码/解码器驱动程序：位于Linux内核空间，通过Linux内核调用驱动程序，调用的驱动程序通常是非标准的驱动程序。

(2) OpenMAX IL层：根据OpenMAX IL层的标准头文件实现不同功能的组件。

另外，Android还提供了OpenMAX的适配层接口，可以对OpenMAX IL的标准组件进行封装并适配。此接口作为Android本地层的接口，可以被Android的多媒体引擎随时调用。

11.2.2 实现OpenMAX IL层接口

在Android系统中，主要使用了OpenMAX的编码/解码器功能，这些功能主要是通过OpenMAX IL层的接口来实现的。

在本小节的内容中，将详细讲解实现OpenMAX IL层接口的基本知识。

1. OpenMAX IL层的接口

(1)　头文件

在 OpenMAX IL 层的接口中定义了若干个头文件，被保存在 frameworks\native\include\media\openmax\目录中。在这些文件中定义了实现 OpenMAX IL 层接口的内容，这些头文件的具体说明如下所示。

- OMX_Types.h：OpenMAX IL 的数据类型定义。
- OMX_Core.h：OpenMAX IL 核心的 API。
- OMX_Component.h：OpenMAX IL 组件相关的 API。
- OMX_Audio.h：音频相关的常量和数据结构。
- OMX_IVCommon.h：图像和视频相关的公共的常量和数据结构。
- OMX_Image.h：图像相关的常量和数据结构。
- OMX_Video.h：视频相关的常量和数据结构。
- OMX_Other.h：其他数据结构(包括 A/V 同步)。
- OMX_Index.h：OpenMAX IL 定义的数据结构索引。
- OMX_ContentPipe.h：内容的管道定义。

在 OpenMAX 标准中只有头文件，没有标准的库。

(2)　实现过程

在具体实现 OpenMAX IL 层的接口时，程序员主要实现包含函数指针的结构体，下面来看在上述头文件中的实现流程。

①　在文件 frameworks\native\include\media\openmax\OMX_Component.h 中定义的 OMX_COMPONENTTYPE 结构体是 OpenMAX IL 层的核心内容，表示一个组件，实现代码如下：

```
typedef struct OMX_COMPONENTTYPE  {
    OMX_U32 nSize;                             /* 定义此结构体的大小 */
    OMX_VERSIONTYPE nVersion;                      /* 版本号 */
    OMX_PTR pComponentPrivate;                 /* 此组件的私有数据指针 */
    /* 调用者(IL client)设置的指针，用于保存它的私有数据，传回给所有的回调函数 */
    OMX_PTR pApplicationPrivate;
    /* 下面的函数指针返回 OMX_core.h 中的对应内容 */
    OMX_ERRORTYPE(*GetComponentVersion)(       /* 获得组件的版本*/
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STRING pComponentName,
            OMX_OUT OMX_VERSIONTYPE *pComponentVersion,
            OMX_OUT OMX_VERSIONTYPE *pSpecVersion,
            OMX_OUT OMX_UUIDTYPE *pComponentUUID);
    OMX_ERRORTYPE(*SendCommand)(               /* 发送命令 */
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_COMMANDTYPE Cmd,
            OMX_IN  OMX_U32 nParam1,
            OMX_IN  OMX_PTR pCmdData);
    OMX_ERRORTYPE(*GetParameter)(              /* 获得参数 */
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_INDEXTYPE nParamIndex,
```

```
        OMX_INOUT OMX_PTR pComponentParameterStructure);
OMX_ERRORTYPE(*SetParameter)(                  /* 设置参数 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_IN  OMX_PTR pComponentParameterStructure);
OMX_ERRORTYPE(*GetConfig)(                     /* 获得配置 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_INOUT OMX_PTR pComponentConfigStructure);
OMX_ERRORTYPE(*SetConfig)(          /* 设置配置 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_IN  OMX_PTR pComponentConfigStructure);
OMX_ERRORTYPE(*GetExtensionIndex)(          /* 转换成 OMX 结构的索引 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_STRING cParameterName,
        OMX_OUT OMX_INDEXTYPE *pIndexType);
OMX_ERRORTYPE(*GetState)(                 /* 获得组件当前的状态 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_OUT OMX_STATETYPE *pState);
OMX_ERRORTYPE(*ComponentTunnelRequest)(       /* 用于连接到另一个组件*/
        OMX_IN  OMX_HANDLETYPE hComp,
        OMX_IN  OMX_U32 nPort,
        OMX_IN  OMX_HANDLETYPE hTunneledComp,
        OMX_IN  OMX_U32 nTunneledPort,
        OMX_INOUT  OMX_TUNNELSETUPTYPE *pTunnelSetup);
OMX_ERRORTYPE(*UseBuffer)(                     /* 为某个端口使用 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE **ppBufferHdr,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes,
        OMX_IN OMX_U8* pBuffer);
OMX_ERRORTYPE(*AllocateBuffer)(           /* 在某个端口分配 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE **ppBuffer,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes);
OMX_ERRORTYPE(*FreeBuffer)(          /*将某个端口 Buffer 释放*/
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_U32 nPortIndex,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
OMX_ERRORTYPE(*EmptyThisBuffer)(          /* 让组件消耗此 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
OMX_ERRORTYPE(*FillThisBuffer)(              /* 让组件填充此 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE *pBuffer);
```

```
    OMX_ERRORTYPE (*SetCallbacks) (          /* 设置回调函数 */
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_CALLBACKTYPE *pCallbacks,
            OMX_IN  OMX_PTR pAppData);
    OMX_ERRORTYPE (*ComponentDeInit) (          /* 反初始化组件 */
            OMX_IN  OMX_HANDLETYPE hComponent);
    OMX_ERRORTYPE (*UseEGLImage) (
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE **ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN void *eglImage);
    OMX_ERRORTYPE (*ComponentRoleEnum) (
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_U8 *cRole,
            OMX_IN OMX_U32 nIndex);
} OMX_COMPONENTTYPE;
```

在实现上述 OMX_COMPONENTTYPE 结构体后，其中调用者可以使用的内容就是各个函数指针，而这些函数指针与文件 OMX_core.h 中定义的内容相对应。

例如在文件 OMX_core.h 中定义 OMX_FreeBuffer 的代码如下所示：

```
#define OMX_FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
```

在文件 OMX_core.h 中定义 OMX_FillThisBuffer 的代码如下所示：

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```

②　接下来需要定义组件运行机制。其中 EmptyThisBuffer 和 FillThisBuffer 是驱动组件运行的基本的机制，前者表示让组件消耗缓冲区，表示对应组件输入的内容；后者表示让组件填充缓冲区，表示对应组件输出的内容。其中定义 OMX_EmptyThisBuffer 的代码如下所示：

```
#define OMX_EmptyThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->EmptyThisBuffer(
        hComponent,
        pBuffer)
```

其中定义 FillThisBuffer 的代码如下所示：

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```

③ 然后开始定义与端口相关的缓冲区管理函数，这些函数分别是 UseBuffer、AllocateBuffer、FreeBuffer，对于组件的端口，有些可以自己分配缓冲区，有些可以使用外部的缓冲区，因此有不同的接口对其进行操作。

④ 使用 SendCommand 向组件发送控制类的命令。接口 GetParameter、SetParameter、GetConfig、SetConfig 用于辅助的参数和配置的设置和获取。具体代码如下所示：

```
#define OMX_GetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
#define OMX_SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
```

⑤ 然后使用 ComponentTunnelRequest 实现组件之间的隧道化连接，在此需要指定两个组

件及其相连的端口。

⑥ 接下来，使用 ComponentDeInit 来反初始化组件。在文件 OMX_Component.h 中定义的端口类型为 OMX_PORTDOMAINTYPE 枚举类型，此枚举的定义代码如下所示：

```
typedef enum OMX_PORTDOMAINTYPE {
   OMX_PortDomainAudio,        /* 音频类型端口 */
   OMX_PortDomainVideo,        /* 视频类型端口 */
   OMX_PortDomainImage,        /* 图像类型端口 */
   OMX_PortDomainOther,        /* 其他类型端口 */
   OMX_PortDomainKhronosExtensions = 0x6F000000,
   OMX_PortDomainVendorStartUnused = 0x7F000000,
   OMX_PortDomainMax = 0x7ffffff
} OMX_PORTDOMAINTYPE;
```

在上述代码中，分别定义了音频类型、视频类型和图像类型，至于其他类型，是 OpenMAX IL 层所定义的 4 种端口的类型。

⑦ 使用 OMX_PARAM_PORTDEFINITIONTYPE 类(也在 OMX_Component.h 中定义)来定义端口的具体内容，其实现代码如下所示：

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
      OMX_U32 nSize;                          /* 结构体大小 */
      OMX_VERSIONTYPE nVersion;               /* 版本*/
      OMX_U32 nPortIndex;                     /* 端口号 */
      OMX_DIRTYPE eDir;                       /* 端口的方向 */
      OMX_U32 nBufferCountActual;             /* 为此端口实际分配的 Buffer 的数目 */
      OMX_U32 nBufferCountMin;                /* 此端口最小 Buffer 的数目*/
      OMX_U32 nBufferSize;                    /* 缓冲区的字节数 */
      OMX_BOOL bEnabled;                      /* 是否使能 */
      OMX_BOOL bPopulated;                    /* 是否在填充 */
      OMX_PORTDOMAINTYPE eDomain;             /* 端口的类型 */
      union {                                 /* 端口实际的内容，由类型确定具体结构 */
         OMX_AUDIO_PORTDEFINITIONTYPE audio;
         OMX_VIDEO_PORTDEFINITIONTYPE video;
         OMX_IMAGE_PORTDEFINITIONTYPE image;
         OMX_OTHER_PORTDEFINITIONTYPE other;
      } format;
      OMX_BOOL bBuffersContiguous;
      OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

对于上述代码的具体说明如下所示。

OMX_DIRTYPE：端口的方向，包含如下两种。

- OMX_DirInput：输入。
- OMX_DirOutput：输出。

端口格式的数据结构：使用 format 联合体来表示，具体由如下 4 种不同类型来表示，与端口的类型相对应。

- OMX_AUDIO_PORTDEFINITIONTYPE

- OMX_VIDEO_PORTDEFINITIONTYPE
- OMX_IMAGE_PORTDEFINITIONTYPE
- OMX_OTHER_PORTDEFINITIONTYPE

上述类型分别在头文件 OMX_Audio.h、OMX_Video.h、OMX_Image.h 和 OMX_Other.h 中定义。

OMX_BUFFERHEADERTYPE：表示一个缓冲区的头部结构，在 OMX_Core.h 中定义。

在文件 OMX_Core.h 中定义了枚举类型 OMX_STATETYPE 来表示 OpenMAX 的状态，主要代码如下所示：

```
typedef enum OMX_STATETYPE
{
    OMX_StateInvalid,                    /* 如果组件监测到内部的数据结构被破坏 */
    OMX_StateLoaded,                     /* 如果组件被加载但是没有完成初始化 */
    OMX_StateIdle,                       /* 如果组件初始化完成，准备开始 */
    OMX_StateExecuting,                  /* 如果组件接受了开始命令，正在受理数据 */
    OMX_StatePause,                      /* 如果组件接受暂停命令*/
    OMX_StateWaitForResources,           /* 如果组件正在等待资源 */
    OMX_StateKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_StateVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_StateMax = 0X7FFFFFFF
} OMX_STATETYPE;
```

在文件 OMX_Core.h 中定义了枚举类型 OMX_COMMANDTYPE，此枚举表示对组件的命令类型，主要代码如下所示：

```
typedef enum OMX_COMMANDTYPE
{
    OMX_CommandStateSet,                 /* 改变状态机器 */
    OMX_CommandFlush,                    /* 刷新数据队列 */
    OMX_CommandPortDisable,              /* 禁止端口 */
    OMX_CommandPortEnable,               /* 使能端口 */
    OMX_CommandMarkBuffer,               /* 标记组件或 Buffer 用于观察 */
    OMX_CommandKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_CommandVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_CommandMax = 0X7FFFFFFF
} OMX_COMMANDTYPE;
```

注意： 在 OpenMAX 的函数参数中，经常包含 OMX_IN 和 OMX_OUT 等宏，它们的实际内容为空，只是为了标记参数的方向是输入还是输出。

2. 在OpenMAX IL层中工作

在实现 OpenMAX IL 层时，一般不调用 OpenMAX DL 层，具体实现的内容是各个不同的组件。通常通过以下两个步骤来实现 OpenMAX IL 组件。

(1) 实现组件的初始化函数

包括硬件和 OpenMAX 数据结构的初始化，主要步骤如下所示。

① 初始化函数指针。
② 初始化私有数据结构。
③ 初始化端口。

在实现上述步骤的过程中，可以使用其中的 pComponentPrivate 成员保留本组件的私有数据为上下文，在最后获得填充，完成 OMX_COMPONENTTYPE 类型的结构体。

(2) 实现 OMX_COMPONENTTYPE 类型结构体的各个指针

在此需要实现其中的各个函数指针，当需要用到私有数据的时候，先从 pComponentPrivate 中得到指针，然后转化成实际的数据结构使用。

因为在 OpenMAX IL 层中，经常用到的组件大多数是一个输入和一个输出端口，所以端口定义的是 OpenMAX IL 组件对外部的接口。对于最常用的编解码(Codec)组件来说，通常需要在每个组件的实现过程中调用硬件的编解码接口来实现。在组件的内部处理中可以通过建立线程来处理。在 OpenMAX 组件的端口中有默认参数，但也可以在运行时设置，因此一个端口也可以支持不同的编码格式。音频编码组件的输出和音频编码组件的输入通常是原始数据格式(PCM 格式)，视频编码组件的输出和视频编码组件的输入通常是原始数据格式(YUV 格式)。

3. OpenMAX适配层

Android 系统中的 OpenMAX 适配层的接口在如下文件中定义：

```
frameworks/av/include/media/IOMX.h
```

文件 IOMX.h 的主要代码如下所示：

```
class IOMX : public IInterface {
public:
    DECLARE_META_INTERFACE(OMX);
    typedef void *buffer_id;
    typedef void *node id;
    virtual bool livesLocally(pid_t pid) = 0;
    struct ComponentInfo {                    // 组件的信息
        String8 mName;
        List<String8> mRoles;
    };
    virtual status_t listNodes(List<ComponentInfo> *list) = 0;  // 节点列表
    virtual status_t allocateNode(
            const char *name, const sp<IOMXObserver> &observer,  // 分配节点
            node_id *node) = 0;
    virtual status_t freeNode(node_id node) = 0;  // 找到节点
    virtual status_t sendCommand(             // 发送命令
            node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param) = 0;
    virtual status_t getParameter(            // 获得参数
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size) = 0;
    virtual status_t setParameter(            // 设置参数
            node_id node, OMX_INDEXTYPE index,
            const void *params, size_t size) = 0;
    virtual status_t getConfig(               // 获得配置
```

```
        node_id node, OMX_INDEXTYPE index,
        void *params, size_t size) = 0;
    virtual status_t setConfig(      // 设置配置
        node_id node, OMX_INDEXTYPE index,
        const void *params, size_t size) = 0;
    virtual status_t useBuffer(      // 使用缓冲区
        node_id node, OMX_U32 port_index, const sp<IMemory> &params,
        buffer_id *buffer) = 0;
    virtual status_t allocateBuffer( // 分配缓冲区
        node_id node, OMX_U32 port_index, size_t size,
        buffer_id *buffer, void **buffer_data) = 0;
    virtual status_t allocateBufferWithBackup(  // 分配后备缓冲区
        node_id node, OMX_U32 port_index, const sp<IMemory> &params,
        buffer_id *buffer) = 0;
    virtual status_t freeBuffer(  // 释放缓冲区
        node_id node, OMX_U32 port_index,buffer_id buffer) = 0;
    virtual status_t fillBuffer(node_id node, buffer_id buffer) = 0; // 填充缓冲区
    virtual status_t emptyBuffer(  // 消耗缓冲区
        node_id node,
        buffer_id buffer,
        OMX_U32 range_offset, OMX_U32 range_length,
        OMX_U32 flags, OMX_TICKS timestamp) = 0;
    virtual status_t getExtensionIndex(
        node_id node,
        const char *parameter_name,
        OMX_INDEXTYPE *index) = 0;
    virtual sp<IOMXRenderer> createRenderer( // 创建渲染器(从 ISurface)
        const sp<ISurface> &surface,
        const char *componentName,
        OMX_COLOR_FORMATTYPE colorFormat,
        size_t encodedWidth, size_t encodedHeight,
        size_t displayWidth, size_t displayHeight) = 0;
    sp<IOMXRenderer> createRenderer(    // 创建渲染器(从 Surface)
        const sp<Surface> &surface,
        const char *componentName,
        OMX_COLOR_FORMATTYPE colorFormat,
        size_t encodedWidth, size_t encodedHeight,
        size_t displayWidth, size_t displayHeight);
    sp<IOMXRenderer> createRendererFromJavaSurface(  // 从 Java 层创建渲染器
        JNIEnv *env, jobject javaSurface,
        const char *componentName,
        OMX_COLOR_FORMATTYPE colorFormat,
        size_t encodedWidth, size_t encodedHeight,
        size_t displayWidth, size_t displayHeight);
};
```

在 IOMX 中，只有第一个 createRenderer 函数是纯虚函数，第二个函数 createRenderer 和函数 createRendererFromJavaSurface 通过调用第一个 createRenderer 函数来实现。

类 IOMXRenderer 表示了一个 OpenMAX 的渲染器，定义此类的代码如下所示：

```
class IOMXRenderer : public IInterface {
public:
    DECLARE_META_INTERFACE(OMXRenderer);
    virtual void render(IOMX::buffer_id buffer) = 0;  // 渲染输出函数
};
```

在类 IOMXRenderer 中只包含了一个 Render 接口，其参数类型 IOMX::buffer_id 其实是 void*，可以根据不同的渲染器使用不同的类型。

在文件 IOMX.h 中还存在一个观察器类 IOMXObserver，此类表示 OpenMAX 的观察者，在里面包含了函数 onMessage()，其参数是 omx_message 结构体。

11.3　分析OpenCore框架

在本节的内容中，将详细讲解 Android 系统中 OpenCore 框架的基本知识，分别介绍其具体结构和相关插件的使用机制，为读者步入本书后面知识的学习打下基础。

11.3.1　OpenCore的层次结构

在 Android 系统中，OpenCore 的另外一个常用的称呼是 PacketVideo，是 Android 多媒体系统的核心。其实 PacketVideo 是一家公司的名称，而 OpenCore 是这套多媒体框架的软件层的名称。在我们 Android 开发者的眼中，二者的含义基本相同。与其他 Android 程序库相比，OpenCore 的代码非常庞大，是基于 C++实现的，定义了全功能的操作系统移植层，各种基本功能均被封装成类的形式，各层次之间的接口使用继承等方式实现。

Android 系统中的 OpenCore 是一个多媒体的框架，从宏观上来看，主要包含了如下两方面的内容。

- PVPlayer：提供了媒体播放器的功能，可以完成各种音频(Audio)、视频(Video)流的回放(Playback)功能。
- PVAuthor：提供了媒体流的记录功能，可以完成各种音频(Audio)、视频(Video)以及静态图像捕获功能。

PVPlayer 和 PVAuthor 以 SDK 的形式提供给开发者，可以在这个 SDK 之上构建多种应用程序和服务。在移动终端中常常使用的多媒体应用程序有媒体播放器、照相机、录像机、录音机等。

OpenCore 系统的基本结构如图 11-9 所示。

在图 11-9 给出的结构中，主要层次元素的具体说明如下所示。

(1) OSCL：OSCL 是 Operating System Compatibility Library 的缩写，意为操作系统兼容库。在 OSCL 中包含了一些操作系统底层的操作，目的是为了更好地在不同操作系统中移植。在 OSCL 中包含的系统底层操作有基本数据类型、配置、字符串工具、I/O、错误处理、线程等内容，类似于一个基础的 C++库。

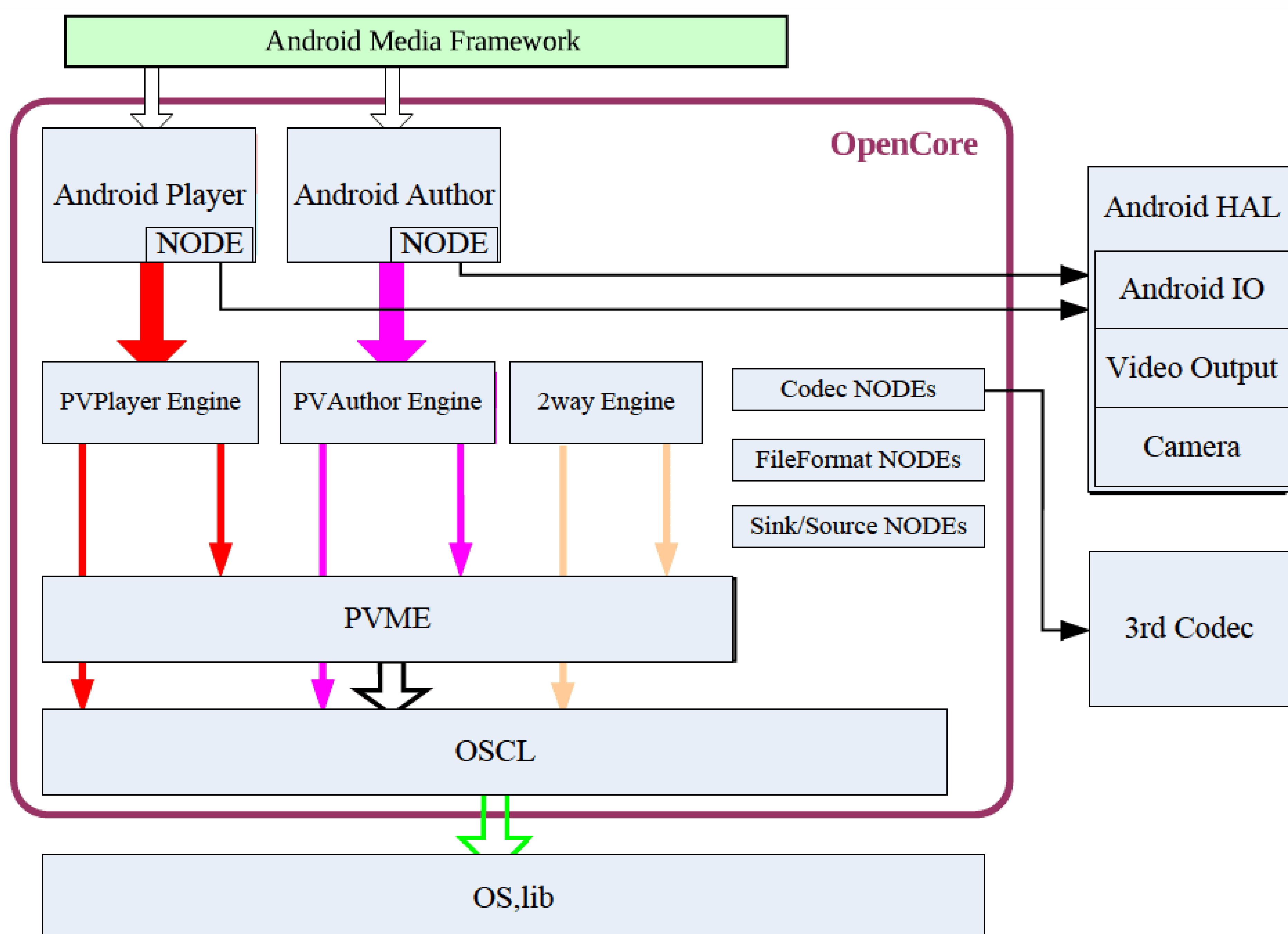


图 11-9　OpenCore的层次结构

(2) PVMF：PVMF 是 PacketVideo Multimedia Framework 的缩写，意为 PV 多媒体框架。PVMF 可以在框架内实现一个文件解析(parser)和组成(composer)、编解码的 Node(节点)，也可以继承其通用的接口，在用户层实现一些 Node。

(3) PVPlayer Engine：PVPlayer Engine 就是 PVPlayer 引擎。

(4) PVAuthor Engine：PVAuthor Engine 就是 PVAuthor 引擎。

除了上述 4 个元素外，其实，在 OpenCore 中包含的内容还有很多。从播放的角度看，PVPlayer 输入的(Source)是文件或者网络媒体流，输出(Sink)的是音频/视频的输出设备，其基本功能包含了媒体流控制、文件解析、音频/视频流的解码(Decode)等方面的内容。除了从文件中播放媒体文件之外，还包含了与网络相关的 RTSP(Real Time Stream Protocol，实时流协议)流。在媒体流记录方面，PVAuthor 的输入(Source)是照相机、麦克风等设备，输出(Sink)是各种文件，包含了流的同步、音频/视频流的编码(Encode)以及文件的写入等功能。

在使用 OpenCore SDK 的时候，有可能需要在应用层实现一个适配器(Adaptor)，然后在适配器之上实现具体的功能，对于 PVMF 的 Node，也可以基于通用的接口，在上层实现，以插件的形式使用。

11.3.2　OpenCore的代码结构

在 Android 系统中，OpenCore 的代码保存在 external/opencore/目录中，此目录是 OpenCore 的根目录，其中包含的各个子目录的具体说明如下所示。

(1) android：是一个上层库，基于 PVPlayer 和 PVAuthor 的 SDK 实现了一个为 Android 使用的 Player 和 Author。

(2) baselibs：在里面包含了数据结构和线程安全等内容的底层库。
(3) codecs_v2：是一个内容较多的库，主要包含了编/解码的实现和 OpenMAX 的实现。
(4) engines：包含 PVPlayer 和 PVAuthor 引擎的实现。
(5) extern_libs_v2：包含了 khronos 的 OpenMAX 的头文件。
(6) fileformats：文件格式的解析(parser)工具。
(7) nodes：在里面提供了 PVMF 的 Node，主要是编解码和文件解析方面的 Node。
(8) oscl：是操作系统兼容库。
(9) pvmi：包含了输入输出控制的抽象接口。
(10) protocols：主要包含了与网络相关的 RTSP、RTP、HTTP 等协议的内容。
(11) pvcommon：是 pvcommon 库文件的 Android.mk 文件，没有源文件。
(12) pvplayer：是 pvplayer 库文件的 Android.mk 文件，没有源文件。
(13) pvauthor：pvauthor 库文件的 Android.mk 文件，没有源文件。
(14) tools_v2：包含了编译工具以及一些可注册的模块。

另外，在 external/opencore/目录中还包含了如下两个文件。

- Android.mk：全局的编译文件。
- pvplayer.conf：配置文件。

在 external/opencore/的各个子文件夹中还包含了很多个 Android.mk 文件，在这些文件之间存在着“递归”的关系。例如，在根目录下的 Android.mk 中包含了下面的内容片段：

```
include $(PV_TOP)/pvcommon/Android.mk
include $(PV_TOP)/pvplayer/Android.mk
include $(PV_TOP)/pvauthor/Android.mk
```

这表示要引用 pvcommon、pvplayer 和 pvauthor 等目录下面的 Android.mk 文件。

external/opencore/目录中的各个 Android.mk 文件可以按照排列组合进行使用，可以将几个 Android.mk 内容合并在一个库里面。

11.3.3　OpenCore的编译结构

(1) 在 Android 开源系统中，通过 OpenCore 编译出来的各个库的具体说明如下所示。

- libopencoreauthor.so：OpenCore 的 Author 库。
- libopencorecommon.so：OpenCore 底层的公共库。
- libopencoredownloadreg.so：下载注册库。
- libopencoredownload.so：下载功能实现库。
- libopencoremp4reg.so：MP4 注册库。
- libopencoremp11.so：MP4 功能实现库。
- libopencorenet_support.so：网络支持库。
- libopencoreplayer.so：OpenCore 的 Player 库。
- libopencorertspreg.so：RTSP 注册库。
- libopencorertsp.so：RTSP 功能实现库。

(2) OpenCore 中的各个库之间的关系如下所示。

- libopencorecommon.so：是所有库的依赖库，提供了公共的功能。
- libopencoreplayer.so 和 libopencoreauthor.so：是两个并立的库，分别用于回放和记录，而且这两个库是 OpenCore 对外的接口库。
- libopencorenet_support.so：提供网络支持的功能。

除此之外，还有一些功能以插件(Plug-In)的方式放入 Player 中使用，每个功能使用两个库，一个实现具体功能，一个用于注册。在接下来的内容中，将简要介绍 OpenCore 中各个库的基本结构。

1. 库libopencorecommon.so的结构

库 libopencorecommon.so 是整个 OpenCore 的核心库，其编译控制的文件路径如下所示：

```
pvcommon/Android.mk
```

上述文件使用递归的方式寻找子文件，其主要内容如下所示：

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//oscl/oscl/osclbase/Android.mk
include $(PV_TOP)//oscl/oscl/osclerror/Android.mk
include $(PV_TOP)//oscl/oscl/osclmemory/Android.mk
include $(PV_TOP)//oscl/oscl/osclutil/Android.mk
include $(PV_TOP)//oscl/pvlogger/Android.mk
include $(PV TOP)//oscl/oscl/osclproc/Android.mk
include $(PV_TOP)//oscl/oscl/osclio/Android.mk
include $(PV_TOP)//oscl/oscl/osclregcli/Android.mk
include $(PV_TOP)//oscl/oscl/osclregserv/Android.mk
include $(PV_TOP)//oscl/unit_test/Android.mk
include $(PV_TOP)//oscl/oscl/oscllib/Android.mk
include $(PV_TOP)//pvmi/pvmf/Android.mk
include $(PV_TOP)//baselibs/pv_mime_utils/Android.mk
include $(PV_TOP)//nodes/pvfileoutputnode/Android.mk
include $(PV_TOP)//baselibs/media_data_structures/Android.mk
include $(PV_TOP)//baselibs/threadsafe_callback_ao/Android.mk
include $(PV_TOP)//codecs_v2/utilities/colorconvert/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/common/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/common/Android.mk
```

这些被包含的 Android.mk 文件真正指定需要编译的文件，这些文件在 Android.mk 的目录及子目录中。事实上，在 libopencorecommon.so 库中包含了以下内容：

- OSCL 的所有内容。
- PVMF 框架部分的内容(pvmi/pvmf/Android.mk)。
- 基础库中的一些内容(baselibs)。
- 编解码的一些内容。
- 文件输出的 node(nodes/pvfileoutputnode/Android.mk)。

从库 libopencorecommon.so 的结构可以看出，最终生成库的结构与 OpenCore 的层次关系并非完全重合。在库 libopencorecommon.so 中已经包含了底层 OSCL 的内容、PVMF 的框架以及 Node 和编解码的工具。

2. 库libopencoreplayer.so的结构

库 libopencoreplayer.so 是一个用于实现播放功能的库，其编译控制的文件的路径如下所示：

```
pvplayer/Android.mk
```

上述文件 Android.mk 的主要代码如下所示：

```
include $(BUILD SHARED LIBRARY)
include $(PV_TOP)//engines/player/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/util/getactualaacconfig/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_wb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/common/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/mp3/dec/Android.mk
include $(PV_TOP)//codecs_v2/utilities/m4v_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/utilities/pv_video_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_common/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_queue/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_h264/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_amr/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_mp3/Android.mk
include $(PV_TOP)//codecs_v2/omx/factories/omx_m4v_factory/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_proxy/Android.mk
include $(PV TOP)//nodes/common/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/oma1/passthru/Android.mk
include $(PV TOP)//pvmi/content policy manager/plugins/common/Android.mk
include $(PV_TOP)//pvmi/media_io/pvmiofileoutput/Android.mk
include $(PV_TOP)//fileformats/common/parser/Android.mk
include $(PV_TOP)//fileformats/id3parcom/Android.mk
include $(PV_TOP)//fileformats/rawgsmamr/parser/Android.mk
include $(PV TOP)//fileformats/mp3/parser/Android.mk
include $(PV_TOP)//fileformats/mp4/parser/Android.mk
include $(PV_TOP)//fileformats/rawaac/parser/Android.mk
include $(PV_TOP)//fileformats/wav/parser/Android.mk
include $(PV_TOP)//nodes/pvaacffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmp3ffparsernode/Android.mk
include $(PV_TOP)//nodes/pvamrffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmediaoutputnode/Android.mk
include $(PV_TOP)//nodes/pvomxvideodecnode/Android.mk
include $(PV_TOP)//nodes/pvomxaudiodecnode/Android.mk
include $(PV_TOP)//nodes/pvwavffparsernode/Android.mk
include $(PV_TOP)//pvmi/recognizer/Android.mk
include $(PV TOP)//pvmi/recognizer/plugins/pvamrffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvmp3ffrecognizer/Android.mk
```

```
include $(PV_TOP)//pvmi/recognizer/plugins/pvwavffrecognizer/Android.mk
include $(PV_TOP)//engines/common/Android.mk
include $(PV_TOP)//engines/adapters/player/framemetadatautility/Android.mk
include $(PV_TOP)//protocols/rtp_payload_parser/util/Android.mk
include $(PV_TOP)//android/Android.mk
include $(PV_TOP)//android/drm/oma1/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_net_support/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

在库 libopencoreplayer.so 中包含了如下内容：

- 解码工具。
- 文件的解析器(MP4)。
- 解码工具对应的 Node。
- player 的引擎部分(路径是 engines/player/Android.mk)。
- Android 的 player 适配器(路径是 android/Android.mk)。
- 识别工具(路径是 pvmi/recognizer)。
- 编解码工具中的 OpenMAX 部分(路径是 codecs_v2/omx)。
- 对应几个插件 Node 的注册。

库 libopencoreplayer.so 中的内容较多，其中主要为各个文件解析器和解码器，PVPlayer 的核心功能在文件 engines/player/Android.mk 中，而文件 android/Android.mk 的内容比较特殊，它是在 PVPlayer 之上构建的一个为 Android 使用的播放器。

3. 库libopencoreauthor.so的结构

库 libopencoreauthor.so 是实现媒体流记录的功能库，其编译控制文件的路径如下所示：

```
pvauthor/Android.mk
```

上述文件 Android.mk 的主要代码如下所示：

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/author/Android.mk
include $(PV_TOP)//codecs_v2/video/m4v_h263/enc/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/enc/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/enc/Android.mk
include $(PV_TOP)//fileformats/mp4/composer/Android.mk
include $(PV_TOP)//nodes/pvamrencnode/Android.mk
include $(PV_TOP)//nodes/pvmp4ffcomposernode/Android.mk
include $(PV_TOP)//nodes/pvvideoencnode/Android.mk
include $(PV_TOP)//nodes/pvavcencnode/Android.mk
include $(PV_TOP)//nodes/pvmediainputnode/Android.mk
include $(PV_TOP)//android/author/Android.mk
```

在库 libopencoreauthor.so 中包含了如下内容：

- 编码工具，例如视频流 H263、H264，音频流 Amr。
- 文件的组成器，例如 MP4。
- 编码工具对应的 Node。
- 用于媒体输入的 Node(目录是 nodes/pvmediainputnode/Android.m)。
- author 引擎(目录是 engines/author/Android.mk)。
- Android 的 author 适配器(目录是 android/author/Android.mk)。

在库 libopencoreauthor.so 中，其内容主要由各个文件编码器和文件组成器构成，其中 PVAuthor 的核心功能在 engines/author/Android.mk 目录中，而文件 android/author/Android.mk 是在 PVAuthor 之上构建的一个为 Android 使用的媒体记录器。

4. 其他库

除了前面介绍的 4 个库之外，在 OpenCore 中还有另外几个库，具体说明如下所示。

(1) 网络支持库 libopencorenet_support.so，对应的 Android.mk 文件的路径如下所示：

```
tools_v2/build/modules/linux_net_support/core/Android.mk
```

(2) MP4 功能实现库 libopencoremp11.so 和注册库 libopencoremp4reg.so，对应的 Android.mk 文件的路径如下所示：

```
tools_v2/build/modules/linux_mp4/core/Android.mk
tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

(3) RTSP 功能实现库 libopencorertsp.so 和注册库 libopencorertspreg.so，对应的 Android.mk 文件的路径如下所示：

```
tools_v2/build/modules/linux_rtsp/core/Android.mk
tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
```

(4) 下载功能实现库 libopencoredownload.so 和注册库 libopencoredownloadreg.so，对应的 Android.mk 文件的路径如下所示：

```
tools_v2/build/modules/linux_download/core/Android.mk
tools_v2/build/modules/linux_download/node_registry/Android.mk
```

11.3.4　操作系统兼容库

OSCL 是 Operating System Compatibility Library(操作系统兼容库)的缩写，在里面包含了一些不同操作系统中移植层的功能，其代码结构如下所示：

```
oscl/oscl
|-- config                  配置的宏
|-- makefile
|-- makefile.pv
|-- osclbase                包含基本类型、宏以及一些 STL 容器等类似的功能
|-- osclerror               错误处理的功能
|-- osclio                  文件 I/O 和 Socket 等功能
```

```
|-- oscllib                    动态库接口等功能
|-- osclmemory                 内存管理、自动指针等功能
|-- osclproc                   线程、多任务通信等功能
|-- osclregcli                 注册客户端的功能
|-- osclregserv                注册服务器的功能
|-- osclutil                   字符串等基本功能
```

在目录 oscl 中，通常用一个目录表示一个模块。OSCL 对应的功能非常详细，几乎封装了 C 语言中的每一个细节功能，并且提供了 C++接口供上层使用。其实在 OpenCore 中的 PVMF 和 Engine 都在使用 OSCL，整个 OpenCore 的调用者也需要使用 OSCL。

在实现 OSCL 时，简单封装了很多典型的 C 语言函数，例如 osclutil 中与数学相关的功能在 oscl_math.inl 中被定义成了内嵌(inline)的函数，具体代码如下所示：

```
OSCL COND EXPORT REF OSCL INLINE double oscl log(double value)
{
    return (double)log(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log10(double value)
{
    return (double)log10(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_sqrt(double value)
{
    return (double)sqrt(value);
}
```

因为文件 oscl_math.inl 被 oscl_math.h 所包含，所以其结果是与函数 oscl_log 的功能等价的原始函数 log。

OSCL 的具体实现比较复杂，很多 C 语言标准库的句柄都被定义成了 C++类的形式，实现起来会比较繁琐。尽管如此，OSCL 的复杂性不是很高。例如以 oscllib 为例，其代码结构如下所示：

```
oscl/oscl/oscllib/
|-- Android.mk
|-- build
|   `-- make
|       `-- makefile
`-- src
    |-- oscl_library_common.h
    |-- oscl library list.cpp
    |-- oscl_library_list.h
    |-- oscl_shared_lib_interface.h
    |-- oscl_shared_library.cpp
    `-- oscl_shared_library.h
```

其中文件 oscl_shared_library.h 是提供给上层使用的动态库的接口功能，定义的接口代码如下所示：

```
class OsclSharedLibrary {
```

```
public:
       OSCL_IMPORT_REF OsclSharedLibrary();
       OSCL_IMPORT_REF OsclSharedLibrary(const OSCL_String& aPath);
       OSCL_IMPORT_REF ~OsclSharedLibrary();
       OSCL_IMPORT_REF OsclLibStatus LoadLib(const OSCL_String& aPath);
       OSCL_IMPORT_REF OsclLibStatus LoadLib();
       OSCL_IMPORT_REF void SetLibPath(const OSCL_String& aPath);
       OSCL_IMPORT_REF OsclLibStatus QueryInterface(
         const OsclUuid &aInterfaceId, OsclAny *&aInterfacePtr);
       OSCL_IMPORT_REF OsclLibStatus Close();
       OSCL_IMPORT_REF void AddRef();
       OSCL_IMPORT_REF void RemoveRef();
}
```

这些接口都与库的加载有关系，而在文件 oscl_shared_library.cpp 中，其具体的功能通过使用函数 dlopen 等来实现。

11.3.5 实现OpenCore中的OpenMAX部分

在 OpenCore 框架中，OpenMAX 是作为插件来实现的，只要封装了 OpenMAX，就可以在 OpenCore 中使用标准的 OpenMAX。

1. OpenMAX的结构

在 OpenCore 中，在如下目录的头文件中包含标准的 OpenMAX：

```
extern_libs_v2/khronos/openmax/include/
```

在文件 build_config/opencore_dynamic/Android_omx_aacdec_sharedlibrary.mk 中，声明了插件 OpenMAX 的主要库是 libomx_sharedlibrary.so，主要代码如下所示：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_WHOLE_STATIC_LIBRARIES := \
    libomx_aac_component_lib \
    libpv_aac_dec
LOCAL_MODULE := libomx_aacdec_sharedlibrary
-include $(PV_TOP)/Android_platform_extras.mk
-include $(PV_TOP)/Android_system_extras.mk

LOCAL_SHARED_LIBRARIES +=  libomx_sharedlibrary libopencore_common

include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)/codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)/codecs_v2/audio/aac/dec/Android.mk
```

库 libomx_sharedlibrary.so 为 omx 针对 OpenCore 的接口层库，也就是说，在每个模拟器上 libomx_sharedlibrary.so 向外(即 OpenCore)提供的接口是一致的。此库可以动态地打开各个 OpenMAX 的编码/解码模块，各个编码/解码模块通过调用 codecs_v2 中 audio 和 video 目录中

软件的编码/解码库来实现。

在 opencore 根目录中，有一个名为 pvplayer.cfg 的文件，此文件用于实现 OpenCore 运行过程的动态配置，文件的主要代码如下所示：

```
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),
 "libopencore_rtspreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),
 "libopencore_downloadreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),
 "libopencore_mp4localreg.so"
(0x6d3413a0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),
 "libopencore mp4localreg.so"
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f),
 "libomx_sharedlibrary.so"
```

2. OpenMAX的接口

在 OpenCore 中，OpenMAX 接口是通过封装标准的 OpenMAX IL 层来构建的，这些接口的基本内容相同，但是不同于标准的 OpenMAX IL 层的 C 语言接口。在 OpenCore 中与 OpenMAX 接口相关的头文件如下所示：

- opencore/codecs_v2/omx/omx_mastercore/include/omx_interface.h：定义插件接口。
- opencore/codecs_v2/omx/omx_common/include/pv_omxcore.h：核心定义。
- opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h：该头文件定义 PV 的 OpenMAX 组件。

文件 omx_interface.h 定义了 OpenMAX 接口的核心功能，在里面包含了各种函数指针的定义类型，具体实现代码如下所示：

```
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Init)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Deinit)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_ComponentNameEnum)(
    OMX_OUT OMX_STRING cComponentName,
    OMX_IN  OMX_U32 nNameLength,
    OMX_IN  OMX_U32 nIndex);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_GetHandle)(
    OMX_OUT OMX_HANDLETYPE *pHandle,
    OMX_IN  OMX_STRING cComponentName,
    OMX_IN  OMX_PTR pAppData,
    OMX_IN  OMX_CALLBACKTYPE *pCallBacks);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_FreeHandle)(
    OMX_IN  OMX_HANDLETYPE hComponent);
typedef OMX_ERRORTYPE(*tpOMX GetComponentsOfRole)(
    OMX_IN  OMX_STRING role,
    OMX_INOUT   OMX_U32 *pNumComps,
    OMX_INOUT   OMX_U8  **compNames);
typedef OMX_ERRORTYPE(*tpOMX_GetRolesOfComponent)(
    OMX_IN   OMX_STRING compName,
    OMX_INOUT   OMX_U32 *pNumRoles,
```

```
    OMX_OUT  OMX_U8 **roles);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_SetupTunnel)(
    OMX_IN  OMX_HANDLETYPE hOutput,
    OMX_IN  OMX_U32 nPortOutput,
    OMX_IN  OMX_HANDLETYPE hInput,
    OMX_IN  OMX_U32 nPortInput);
typedef OMX_ERRORTYPE(*tpOMX_GetContentPipe)(
    OMX_OUT OMX_HANDLETYPE *hPipe,
    OMX_IN OMX_STRING szURI);
typedef OMX_BOOL(*tpOMXConfigParser)(
    OMX_PTR aInputParameters,
    OMX_PTR aOutputParameters);
```

上述函数指针是 OpenMAX 的核心方法，这些指针类型需要使用继承来设置。

另外，在文件 omx_interface.h 中还定义了类 OMXInterface，在类中包含了一系列函数，这些函数返回的都是上面类型的函数指针。

类 OMXInterface 是 OpenMAX 直接实现 OpenCore 的接口：

```
class OMXInterface : public OsclSharedLibraryInterface
{
    public:
        OMXInterface()
        {
            pOMX_Init = NULL;
            pOMX_Deinit = NULL;
            pOMX_ComponentNameEnum = NULL;
            pOMX_GetHandle = NULL;
            pOMX_FreeHandle = NULL;
            pOMX_GetComponentsOfRole = NULL;
            pOMX_GetRolesOfComponent = NULL;
            pOMX_SetupTunnel = NULL;
            pOMX_GetContentPipe = NULL;
            pOMXConfigParser = NULL;
        };
        virtual bool UnloadWhenNotUsed(void) = 0;
        tpOMX_Init GetpOMX_Init()
        {
            return pOMX_Init;
        };
        tpOMX_Deinit GetpOMX_Deinit()
        {
            return  pOMX_Deinit;
        };
        tpOMX_ComponentNameEnum GetpOMX_ComponentNameEnum()
        {
            return pOMX_ComponentNameEnum;
        };
        tpOMX_GetHandle GetpOMX_GetHandle()
        {
```

```
        return pOMX_GetHandle;
    };
    tpOMX_FreeHandle GetpOMX_FreeHandle()
    {
        return pOMX_FreeHandle;
    };
    tpOMX_GetComponentsOfRole GetpOMX_GetComponentsOfRole()
    {
        return pOMX_GetComponentsOfRole;
    };
    tpOMX_GetRolesOfComponent GetpOMX_GetRolesOfComponent()
    {
        return pOMX_GetRolesOfComponent;
    };
    tpOMX_SetupTunnel GetpOMX_SetupTunnel()
    {
        return pOMX_SetupTunnel;
    };
    tpOMX_GetContentPipe GetpOMX_GetContentPipe()
    {
        return pOMX GetContentPipe;
    };
    tpOMXConfigParser GetpOMXConfigParser()
    {
        return pOMXConfigParser;
    };
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Init)(void);
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Deinit)(void);
    OMX ERRORTYPE OMX APIENTRY(*pOMX ComponentNameEnum)(
        OMX_OUT OMX_STRING cComponentName,
        OMX_IN  OMX_U32 nNameLength,
        OMX_IN  OMX_U32 nIndex);
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_GetHandle)(
        OMX_OUT OMX_HANDLETYPE* pHandle,
        OMX_IN  OMX_STRING cComponentName,
        OMX_IN  OMX_PTR pAppData,
        OMX_IN  OMX_CALLBACKTYPE *pCallBacks);
    OMX_ERRORTYPE OMX_APIENTRY(*pOMX_FreeHandle)(
        OMX_IN  OMX_HANDLETYPE hComponent);
    OMX_ERRORTYPE(*pOMX_GetComponentsOfRole)(
        OMX_IN  OMX_STRING role,
        OMX_INOUT   OMX_U32 *pNumComps,
        OMX_INOUT   OMX_U8  **compNames);
    OMX_ERRORTYPE(*pOMX_GetRolesOfComponent)(
        OMX_IN   OMX_STRING compName,
        OMX_INOUT   OMX_U32 *pNumRoles,
        OMX_OUT  OMX_U8 **roles);
    OMX ERRORTYPE OMX APIENTRY(*pOMX SetupTunnel)(
        OMX_IN  OMX_HANDLETYPE hOutput,
```

```
            OMX_IN  OMX_U32 nPortOutput,
            OMX_IN  OMX_HANDLETYPE hInput,
            OMX_IN  OMX_U32 nPortInput);
        OMX_ERRORTYPE(*pOMX_GetContentPipe)(
            OMX_OUT OMX_HANDLETYPE *hPipe,
            OMX_IN OMX_STRING szURI);
        OMX_BOOL(*pOMXConfigParser)(
            OMX_PTR aInputParameters,
            OMX_PTR aOutputParameters);
};
```

3. OpenMAX的组织结构

在文件 opencore/codecs_v2/omx/omx_sharedlibrary/interface/src/pv_omx_interface.cpp 中实现了类 OMXInterface，在实现时是通过实现类里面的函数指针方式来实现的。

在文件 pv_omx_interface.cpp 中，函数 PVGetInterface()和 PVReleaseInterface()是使用 C 语言导出的函数，这两个函数的实现代码如下所示：

```
extern "C"
{
    OSCL_EXPORT_REF OsclAny* PVGetInterface()
    {
        return PVOMXInterface::Instance();
    }
    OSCL_EXPORT_REF void PVReleaseInterface(void *interface)
    {
        PVOMXInterface *pInterface = (PVOMXInterface*)interface;
        if (pInterface)
        {
            OSCL_DELETE(pInterface);
        }
    }
}
```

在文件 pv_omx_interface.cpp 中，类 PVOMXInterface 继承了 OMXInterface，在此类的构造函数中设置了各个 OMXInterface 中的函数指针。

构造函数 PVOMXInterface()的主要代码如下所示：

```
private:
    PVOMXInterface()
    {
        //设置指针 OMX 的核心方法
        pOMX_Init = OMX_Init;
        pOMX_Deinit = OMX_Deinit;
        pOMX_ComponentNameEnum = OMX_ComponentNameEnum;
        pOMX_GetHandle = OMX_GetHandle;
        pOMX_FreeHandle = OMX_FreeHandle;
        pOMX_GetComponentsOfRole = OMX_GetComponentsOfRole;
        pOMX_GetRolesOfComponent = OMX_GetRolesOfComponent;
```

```
        pOMX_SetupTunnel = OMX_SetupTunnel;
        pOMX GetContentPipe = OMX GetContentPipe;
        pOMXConfigParser = OMXConfigParser;
    };
```

我们所介绍的上述构造函数，都是在文件 opencore/codecs_v2/omx/omx_common/src/pv_omxcore.cpp 中实现的，此文件实现了 OpenMAX 的核心功能。

文件 opencore/codecs_v2/omx/omx_common/src/pv_omxregistry.cpp 的功能是注册 OpenMAX 模块，其主要实现代码如下所示：

```
//注册 MP3 解码器
OMX_ERRORTYPE Mp3Register()
{
    ComponentRegistrationType *pCRT = (ComponentRegistrationType *)
      oscl_malloc(sizeof(ComponentRegistrationType));
    if (pCRT)
    {
       pCRT->ComponentName = (OMX_STRING)"OMX.PV.mp3dec"; //组件名
       pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.mp3";
       pCRT->NumberOfRolesSupported = 1;
       pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
       pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
       pCRT->FunctionPtrDestroyComponent =
         &OmxComponentFactoryDynamicDestructor;
       pCRT->SharedLibraryName = (OMX_STRING)"libomx_mp3dec_sharedlibrary.so";
       pCRT->SharedLibraryPtr = NULL;
       OsclUuid *temp = (OsclUuid*)oscl_malloc(sizeof(OsclUuid));
       if (temp == NULL)
       {
          oscl_free(pCRT); // 释放内存
          return OMX_ErrorInsufficientResources;
       }
       OSCL_PLACEMENT_NEW(temp, PV_OMX_MP3DEC_UUID);
       pCRT->SharedLibraryOsclUuid = (OMX_PTR)temp;
       pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_MP3_COMPONENT
#if (DYNAMIC_LOAD_OMX_MP3_COMPONENT == 0)
       pCRT->FunctionPtrCreateComponent = &Mp3OmxComponentFactory;
       pCRT->FunctionPtrDestroyComponent = &Mp3OmxComponentDestructor;
       pCRT->SharedLibraryName = NULL;
       pCRT->SharedLibraryPtr = NULL;
       if (pCRT->SharedLibraryOsclUuid)
          oscl_free(pCRT->SharedLibraryOsclUuid);
       pCRT->SharedLibraryOsclUuid = NULL;
       pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
```

```
    }
    else
    {
        return OMX_ErrorInsufficientResources;
    }
    return  ComponentRegister(pCRT);
}
//WMA 格式解码
OMX_ERRORTYPE WmaRegister()
{
    ComponentRegistrationType *pCRT = (ComponentRegistrationType*)
      oscl_malloc(sizeof(ComponentRegistrationType));
    if (pCRT)
    {
        pCRT->ComponentName = (OMX_STRING)"OMX.PV.wmadec";
        pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.wma";
        pCRT->NumberOfRolesSupported = 1;
        pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
        pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
        pCRT->FunctionPtrDestroyComponent =
          &OmxComponentFactoryDynamicDestructor;
        pCRT->SharedLibraryName = (OMX_STRING)"libomx_wmadec_sharedlibrary.so";
        pCRT->SharedLibraryPtr = NULL;
        OsclUuid *temp = (OsclUuid*)oscl_malloc(sizeof(OsclUuid));
        if (temp == NULL)
        {
            oscl_free(pCRT); // free allocated memory
            return OMX_ErrorInsufficientResources;
        }
        OSCL_PLACEMENT_NEW(temp, PV_OMX_WMADEC_UUID);
        pCRT->SharedLibraryOsclUuid = (OMX_PTR)temp;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_WMA_COMPONENT
#if (DYNAMIC_LOAD_OMX_WMA_COMPONENT == 0)
        pCRT->FunctionPtrCreateComponent = &WmaOmxComponentFactory;
        pCRT->FunctionPtrDestroyComponent = &WmaOmxComponentDestructor;
        pCRT->SharedLibraryName = NULL;
        pCRT->SharedLibraryPtr = NULL;
        if (pCRT->SharedLibraryOsclUuid)
            oscl_free(pCRT->SharedLibraryOsclUuid);
        pCRT->SharedLibraryOsclUuid = NULL;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
    }
    else
    {
```

```
        return OMX_ErrorInsufficientResources;
    }
    return  ComponentRegister(pCRT);
}
```

4. 实现OpenMAX的编码/解码组件

OpenMAX 的主要功能是通过解码/编码组件来实现的，各个组件的基本结构类似，它们的实现内容实际上就是文件 opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h 中定义的类 OmxComponentBase。假如要实现 MP3 格式文件的解码处理，则在如下目录中实现了 MP3 的解码功能：

```
opencore/codecs_v2/omx/mp3
```

在上述目录中，文件 Android.mk 生成了名为 libomx_mp3_component_lib.so 的库，此静态库将被连接，生成动态库 libomx_mp3dec_sharedlibrary_lib。此 Android.mk 文件的主要代码如下所示：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
    src/mp3_dec.cpp \
    src/omx_mp3_component.cpp \
    src/mp3_timestamp.cpp
LOCAL_MODULE := libomx_mp3_component_lib
LOCAL_CFLAGS :=  $(PV_CFLAGS)
LOCAL_ARM_MODE := arm
LOCAL_STATIC_LIBRARIES :=
LOCAL_SHARED_LIBRARIES :=
LOCAL_C_INCLUDES := \
    $(PV_TOP)/codecs_v2/omx/omx_mp3/src \
    $(PV_TOP)/codecs_v2/omx/omx_mp3/include \
    $(PV_TOP)/extern_libs_v2/khronos/openmax/include \
    $(PV_TOP)/codecs_v2/omx/omx_baseclass/include \
    $(PV_TOP)/codecs_v2/audio/mp3/dec/src \
    $(PV_TOP)/codecs_v2/audio/mp3/dec/include \
    $(PV_INCLUDES)
LOCAL_COPY_HEADERS_TO  := $(PV_COPY_HEADERS_TO)
LOCAL_COPY_HEADERS := \
    include/mp3_dec.h \
    include/omx_mp3_component.h \
    include/mp3_timestamp.h
include $(BUILD_STATIC_LIBRARY)
```

在目录 opencore/codecs_v2/omx/omx_mp3/src/中存在如下 3 个文件。

- mp3_dec.cpp：能够调用 MP3 解码器组件。
- mp3_timestamp.cpp：能够实现时间戳功能。
- omx_mp3_component.cpp：定义了 MP3 解码器组件。

在文件 opencore/codecs_v2/omx/omx_mp3/include/omx_mp3_component.h 中，定义了类 OpenmaxMp3AO，此类继承了 OmxComponentAudio，主要代码如下所示：

```
class OpenmaxMp3AO : public OmxComponentAudio {
   public:
      OpenmaxMp3AO();
      ~OpenmaxMp3AO();
      OMX_ERRORTYPE ConstructComponent(OMX_PTR pAppData, OMX_PTR pProxy);
      OMX_ERRORTYPE DestroyComponent();
      OMX_ERRORTYPE ComponentInit();
      OMX_ERRORTYPE ComponentDeInit();
      static void ComponentGetRolesOfComponent(OMX_STRING *aRoleString);
      void ProcessData();
      void SyncWithInputTimestamp();
      void ProcessInBufferFlag();
      void ResetComponent();
      OMX_ERRORTYPE GetConfig(
         OMX_IN  OMX_HANDLETYPE hComponent,
         OMX_IN  OMX_INDEXTYPE nIndex,
         OMX_INOUT OMX_PTR pComponentConfigStructure);
   private:
      void CheckForSilenceInsertion();
      void DoSilenceInsertion();
      Mp3Decoder *ipMp3Dec;
      Mp3TimeStampCalc iCurrentFrameTS;
};
```

在文件 omx_mp3_component.cpp 中定义了 MP3 解码器组件，通过函数 ProcessData()实现 MP3 文件的解码处理。函数 ProcessData()的实现代码如下所示：

```
void OpenmaxMp3AO::ProcessData()
{
    PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
      (0, "OpenmaxMp3AO : ProcessData IN"));

   QueueType *pInputQueue  = ipPorts[OMX_PORT_INPUTPORT_INDEX]->pBufferQueue;
   QueueType *pOutputQueue = ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->pBufferQueue;

   ComponentPortType *pInPort = (ComponentPortType*)ipPorts[OMX_PORT_INPUTPORT_INDEX];
   ComponentPortType *pOutPort = ipPorts[OMX_PORT_OUTPUTPORT_INDEX];
   OMX_COMPONENTTYPE *pHandle = &iOmxComponent;

   OMX_U8 *pOutBuffer; //输出缓冲区的指针
   OMX_U32 OutputLength; //输出缓冲区的长度
   OMX_S32 DecodeReturn;
   OMX_BOOL ResizeNeeded = OMX_FALSE;

    OMX_U32 TempInputBufferSize = (2 * sizeof(uint8) *
      (ipPorts[OMX_PORT_INPUTPORT_INDEX]->PortParam.nBufferSize));
```

```
    if ((!iIsInputBufferEnded) || iEndofStream)
    {
        if (OMX_TRUE == iSilenceInsertionInProgress)
        {
            DoSilenceInsertion();
            //If the flag is still true, come back to this routine again
            if (OMX_TRUE == iSilenceInsertionInProgress)
            {
                return;
            }
        }
        //证实prev是否发布了buffer
        if (OMX_TRUE == iNewOutBufRequired)
        {
            //证实是否一个新的输出缓冲区是可利用的
            if (0 == (GetQueueNumElem(pOutputQueue)))
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
                 (0, "OpenmaxMp3AO : ProcessData OUT output buffer unavailable"));
                return;
            }
            ipOutputBuffer = (OMX_BUFFERHEADERTYPE*)DeQueue(pOutputQueue);
            if (NULL == ipOutputBuffer)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
(0, "OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                return;
            }
            ipOutputBuffer->nFilledLen = 0;
            iNewOutBufRequired = OMX_FALSE;
            //把输出缓冲区时间戳设置为当前时间戳
            ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
            //复制在动态重组之前当地存放的输出缓冲区
            //被接受的新的OMX缓冲
            if (OMX_TRUE == iSendOutBufferAfterPortReconfigFlag)
            {
                if ((ipTempOutBufferForPortReconfig)
                  && (iSizeOutBufferForPortReconfig <= ipOutputBuffer->nAllocLen))
                {
                    oscl_memcpy(ipOutputBuffer->pBuffer,
                      ipTempOutBufferForPortReconfig,
                      iSizeOutBufferForPortReconfig);
                    ipOutputBuffer->nFilledLen = iSizeOutBufferForPortReconfig;
                    ipOutputBuffer->nTimeStamp=iTimestampOutBufferForPortReconfig;
                }
                iSendOutBufferAfterPortReconfigFlag = OMX_FALSE;
                //只有当充满时退还输出缓冲区
                if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen)
                   < iOutputFrameLength)
```

```
            {
                ReturnOutputBuffer(ipOutputBuffer, pOutPort);
            }
            //释放临时输出缓冲区
            if (ipTempOutBufferForPortReconfig)
            {
                oscl_free(ipTempOutBufferForPortReconfig);
                ipTempOutBufferForPortReconfig = NULL;
                iSizeOutBufferForPortReconfig = 0;
            }
            //Dequeue new output buffer if required
            //to continue decoding the next frame
            if (OMX_TRUE == iNewOutBufRequired)
            {
                if (0 == (GetQueueNumElem(pOutputQueue)))
                {
                    PVLOGGER_LOGMSG(
                      PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
                      (0,
                      "OpenmaxMp3AO : ProcessData OUT, output buffer unavailable"));
                    return;
                }
                ipOutputBuffer = (OMX_BUFFERHEADERTYPE*)DeQueue(pOutputQueue);
                if (NULL == ipOutputBuffer)
                {
                    PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
                     (0,
"OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                    return;
                }
                ipOutputBuffer->nFilledLen = 0;
                iNewOutBufRequired = OMX_FALSE;
                ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
            }
        }
    }
    /*标号缓冲的代码
     * 根据 hMarkTargetComponent 设置规格
     */
    if (ipMark != NULL)
    {
        ipOutputBuffer->hMarkTargetComponent = ipMark->hMarkTargetComponent;
        ipOutputBuffer->pMarkData = ipMark->pMarkData;
        ipMark = NULL;
    }
    if (ipTargetComponent != NULL)
    {
        ipOutputBuffer->hMarkTargetComponent = ipTargetComponent;
        ipOutputBuffer->pMarkData = iTargetMarkData;
```

```
        ipTargetComponent = NULL;
    }
    //在此标记缓冲代码末端
    pOutBuffer = &ipOutputBuffer->pBuffer[ipOutputBuffer->nFilledLen];
    OutputLength = 0;
    /*复制临时被存放的前个输入缓冲区的残余数据
     *缓冲接踵而来的数据流
     */
    if (iTempInputBufferLength > 0
      &&((iInputCurrLength + iTempInputBufferLength) < TempInputBufferSize))
    {
        oscl_memcpy(&ipTempInputBuffer[iTempInputBufferLength],
          ipFrameDecodeBuffer, iInputCurrLength);
        iInputCurrLength += iTempInputBufferLength;
        iTempInputBufferLength = 0;
        ipFrameDecodeBuffer = ipTempInputBuffer;
    }
    //将输出缓冲区作为指针
    DecodeReturn = ipMp3Dec->Mp3DecodeAudio(//设置 ipMp3Dec 的类型是 Mp3Decode
                  (OMX_S16*)pOutBuffer, //输出缓冲区的指针
                  (OMX_U32*)&OutputLength, //输出缓冲区的长度
                   &(ipFrameDecodeBuffer),
                   &iInputCurrLength,
                   &iFrameCount,
                   &(ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->AudioPcmMode),
                   &(ipPorts[OMX_PORT_INPUTPORT_INDEX]->AudioMp3Param),
                   iEndOfFrameFlag,
                   &ResizeNeeded);
    if (ResizeNeeded == OMX_TRUE)
    {
        if (0 != OutputLength)
        {
            iOutputFrameLength = OutputLength * 2;
            //更新时间戳
            iSamplesPerFrame =
OutputLength / ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->AudioPcmMode.nChannels;
            iCurrentFrameTS.SetParameters(
              ipPorts[OMX_PORT_OUTPUTPORT_INDEX] ->AudioPcmMode.nSamplingRate,
              iSamplesPerFrame);
            iOutputMilliSecPerFrame = iCurrentFrameTS.GetFrameDuration();
        }
        iResizePending = OMX TRUE;
        /*不要退回引起的输出缓冲区，当地存放它
         *并且等待动态接口重新构造完成*/
        if ((NULL == ipTempOutBufferForPortReconfig))
        {
            ipTempOutBufferForPortReconfig =
```

```
            (OMX_U8*)oscl_malloc(sizeof(uint8) * OutputLength * 2);
        if (NULL == ipTempOutBufferForPortReconfig)
        {
            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
         (0, "OpenmaxMp3AO : ProcessData error, insufficient resources"));
            return;
        }
    }
    //复制 omx 输出缓冲区的临时内部缓冲
    oscl_memcpy(ipTempOutBufferForPortReconfig, pOutBuffer, OutputLength * 2);
    iSizeOutBufferForPortReconfig = OutputLength * 2;
    //设置当前时间戳对第一个产品框架的输出缓冲区时间戳
    //以后将取消
    iTimestampOutBufferForPortReconfig = iCurrentFrameTS.GetConvertedTs();
    iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
    OutputLength = 0;
    OMX_COMPONENTTYPE *pHandle = (OMX_COMPONENTTYPE*)ipAppPriv->CompHandle;
    (*(ipCallbacks->EventHandler))
    (pHandle,
     iCallbackData,
     OMX EventPortSettingsChanged, //The command was completed
     OMX_PORT_OUTPUTPORT_INDEX,
     0,
     NULL);
}
ipOutputBuffer->nFilledLen += OutputLength * 2;
ipOutputBuffer->nOffset = 0;
if (OutputLength > 0)
{
    iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
}
if (OMX_TRUE == iEndofStream)
{
    if (MP3DEC_SUCCESS != DecodeReturn)
    {
        PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
            (0, "OpenmaxMp3AO : ProcessData EOS callback send"));
        (*(ipCallbacks->EventHandler))
        (pHandle,
         iCallbackData,
         OMX_EventBufferFlag,
         1,
         OMX_BUFFERFLAG_EOS,
         NULL);
        iEndofStream = OMX_FALSE;
        ipOutputBuffer->nFlags |= OMX_BUFFERFLAG_EOS;
        ReturnOutputBuffer(ipOutputBuffer, pOutPort);
        ipOutputBuffer = NULL;
        PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
```

```
            (0, "OpenmaxMp3AO : ProcessData OUT"));
          return;
        }
    }
    if (MP3DEC_SUCCESS == DecodeReturn)
    {
        ipInputBuffer->nFilledLen = iInputCurrLength;
    }
    else if (MP3DEC_INCOMPLETE_FRAME == DecodeReturn)
    {
        oscl_memcpy(ipTempInputBuffer, ipFrameDecodeBuffer, iInputCurrLength);
        iTempInputBufferLength = iInputCurrLength;
        ipInputBuffer->nFilledLen = 0;
        iInputCurrLength = 0;
    }
    else
    {
        ipInputBuffer->nFilledLen = 0;
        iInputCurrLength = 0;
        PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
         (0, "OpenmaxMp3AO : ProcessData ErrorStreamCorrupt callback send"));
        (*(ipCallbacks->EventHandler))
        (pHandle,
         iCallbackData,
         OMX_EventError,
         OMX_ErrorStreamCorrupt,
         0,
         NULL);
    }
    //如果它经过译码器处理，则会得到充分的消耗并退回到输入缓冲区
    if (0 == ipInputBuffer->nFilledLen)
    {
        ReturnInputBuffer(ipInputBuffer, pInPort);
        ipInputBuffer = NULL;
        iIsInputBufferEnded = OMX_TRUE;
        iInputCurrLength = 0;
    }
    //当充满时送回输出缓冲区
    if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen)
       < (iOutputFrameLength))
    {
        ReturnOutputBuffer(ipOutputBuffer, pOutPort);
        ipOutputBuffer = NULL;
    }
    /* 如果有些处理在当前缓冲中，则重新编排 AO */
    if (((iInputCurrLength != 0 || GetQueueNumElem(pInputQueue) > 0)
       && (GetQueueNumElem(pOutputQueue) > 0) && (ResizeNeeded == OMX_FALSE))
       || (OMX_TRUE == iEndofStream))
    {
```

```
            RunIfNotReady();
        }
    }
    PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE,
      (0, "OpenmaxMp3AO : ProcessData OUT"));
    return;
}
```

11.3.6　OpenCore扩展详解

在 Android 系统中，除了可以使用 OpenCore 本身提供的强大功能外，还可以对 OpenCore 进行扩展，以实现更加强大的功能。

1. OpenCore Node

在扩展 OpenCore 时，一般是基于 OpenCore 的框架为其增加固定的插件，插件主要做成 Node 的形式。

(1)　其中与编解码相关的 Node 如下所示：

- pvomxbasedecnode
- pvomxaudiodecnode
- pvomxvideodencode
- pvomxencnode

(2)　在扩展 OpenCore 时，与文件格式相关的 Node 如下所示：

- pvwavffparsernode
- Pvaacffparsernode
- Pvamrffparsernode
- pvmp3ffparsernode
- pvmp4ffparsernode
- pvvideoparsernode
- pvmp4f fcomposernode

(3)　在扩展 OpenCore 时，与输入输出相关的 Node 如下所示：

- pvmediainputnode
- pvmediaoutputnode
- pvdummyinputnode
- pvdummyoutputnode
- pvfileoutputnode
- pvdownloadmanagernode

除了上面列出的这些 Node 之外，还包括一些其他功能的常用 Node，例如 pvsocketnode 和 pvdownloadmanagernode 等。

2. MediaIO

MediaIO 的缩写是 MIO，在 opencore/pvmi/pvmf/include/目录中有如下头文件定义：

- pvmiMIOControl.h
- pvmi_media_transfer.h

在实现的过程中，只需要继承和构建其中的接口，然后由框架最终实现成为 Node，在 OpenCore 系统中使用。

其实 MediaIO 是对 Node 的一种封装，将其封装成多媒体的输入输出环节。

在文件 pvmi_mio_control.h 中，定义类 PvmiMIOControl，来表示 MIO 的控制类接口，定义此类的代码如下所示：

```
class PvmiMIOControl{
   public:
      virtual ~PvmiMIOControl() {}
      virtual PVMFStatus connect(PvmiMIOSession& aSession,
                             PvmiMIOObserver* aObserver) = 0;
      virtual PVMFStatus disconnect(PvmiMIOSession aSession) = 0;
      virtual PvmiMediaTransfer* createMediaTransfer(
         PvmiMIOSession &aSession,
         PvmiKvp *read_formats = NULL, int32 read_flags = 0,
         PvmiKvp *write_formats = NULL, int32 write_flags = 0) = 0;
      virtual void deleteMediaTransfer(
         PvmiMIOSession &aSession,
         PvmiMediaTransfer *media_transfer) = 0;
      virtual PVMFCommandId QueryUUID(const PvmfMimeString &aMimeType,
                                Oscl_Vector<PVUuid, OsclMemAllocator> &aUuids,
                                bool aExactUuidsOnly = false,
                                const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId QueryInterface(const PVUuid &aUuid,
                                    PVInterface *&aInterfacePtr,
                                    const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Init(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Reset(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Start(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Pause(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Flush(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId DiscardData(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId DiscardData(
        PVMFTimestamp aTimestamp, const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId Stop(const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId CancelCommand(
        PVMFCommandId aCmd, const OsclAny *aContext = NULL) = 0;
      virtual PVMFCommandId CancelAllCommands(const OsclAny *aContext = NULL) = 0;
      virtual void ThreadLogon() = 0;
      virtual void ThreadLogoff() = 0;
};
```

在上述代码中，有很多函数使用 OsclAny 类型的指针作为参数，这样的好处是可以使用所有数据结构。其中接口 Init()、Reset()、Start()、Pause()、Flush()和 Stop()可实现流控制，而函数 createMediaTransfer 用于得到类 PvmiMediaTransfer。

而在文件 pvmi_media_transfer.h 中，定义类 PvmiMediaTransfer 来表示 MIO 的数据接口，定义此类的代码如下所示：

```
class PvmiMediaTransfer
{
   public:
      virtual ~PvmiMediaTransfer() {}
      virtual void setPeer(PvmiMediaTransfer *aPeer) = 0;
      virtual void useMemoryAllocators(OsclMemAllocator *read_write_alloc) = 0;
      virtual PVMFCommandId writeAsync(
         uint8 format_type, int32 format_index,
         uint8 *data, uint32 data_len,
         const PvmiMediaXferHeader &data_header_info,
         OsclAny *aContext = NULL) = 0;
      virtual void writeComplete(
         PVMFStatus aStatus,
         PVMFCommandId write_cmd_id,
         OsclAny *aContext) = 0;
      virtual PVMFCommandId readAsync(
         uint8 *data, uint32 max_data_len,
         OsclAny *aContext = NULL,
         int32 *formats = NULL, uint16 num_formats = 0) = 0;
      virtual void readComplete(
         PVMFStatus aStatus,
         PVMFCommandId read_cmd_id,
         int32 format_index,
         const PvmiMediaXferHeader &data_header_info,
         OsclAny *aContext) = 0;
      virtual void statusUpdate(uint32 status_flags) = 0;
      virtual void cancelCommand(PVMFCommandId command_id) = 0;
      virtual void cancelAllCommands() = 0;
};
```

3. OpenCore Player

OpenCore 的 Player 的编译文件是 pvplayer/Android.mk，编译后，将会生成动态库文件 libopencoreplayer.so，在此库中包含了如下两方面的内容：

- Player 的 Engine(引擎)。
- 为 Android 构建的 Player，是一个适配器(Adapter)。

Engine 的路径是 engine/player，Adapter 的路径是 android。

在库 libopencoreplayer.so 中还包含了下面的内容：

- 解码工具。
- 文件的解析器。
- 解码工具对应的 Node。
- Player 的引擎部分，编译文件是 engines/player/Android.mk。
- 为 Android 构建的 Player 适配器，编译文件是 android/Android.mk。

- 识别工具，目录是 pvmi/recognizer。
- 编解码工具中的 OpenMAX 部分，目录是 codecs_v2/omx。
- 对应插件 Node 的注册。

由此可见，库 libopencoreplayer.so 中的内容较多，其中主要的功能是针对各个文件解析器和解码器的。

PVPlayer 的核心功能在文件 engines/player/Android.mk 中。而文件 android/Android.mk 的内容比较特殊，功能上是在 PVPlayer 之上构建的一个可以供 Android 使用的播放器。

库 libopencoreplayer.so 的具体结构如图 11-10 所示。

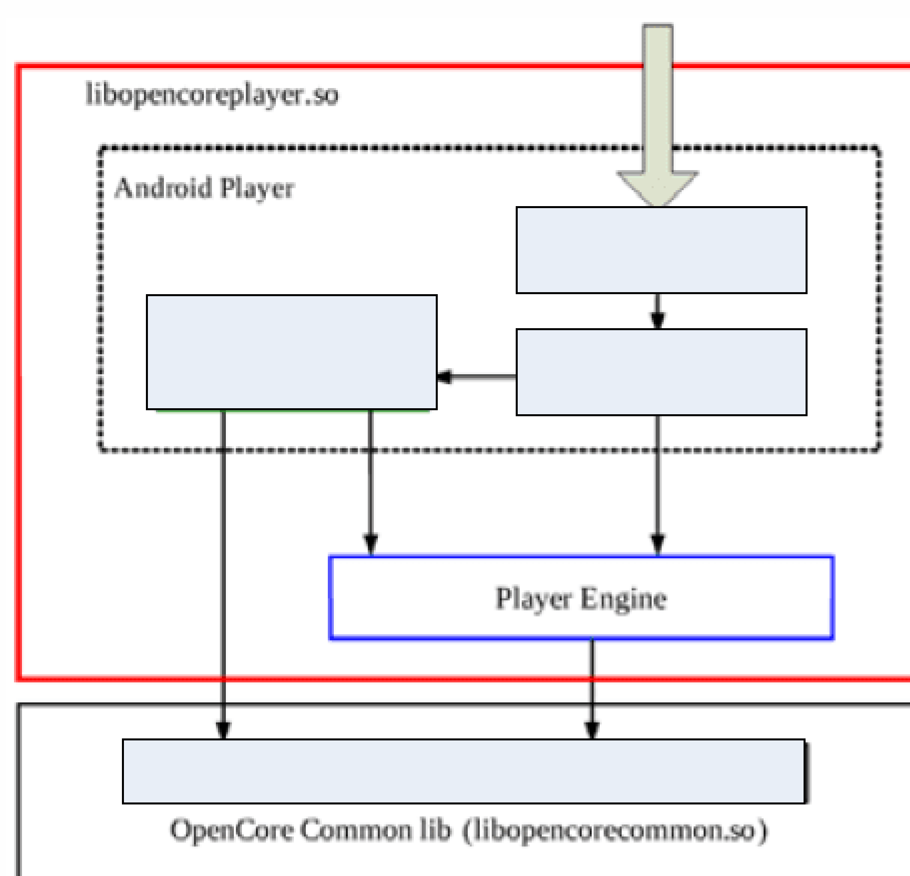


图 11-10　库libopencoreplayer.so的结构

(1)　Player Engine

Player Engine 的类结构如图 11-11 所示。

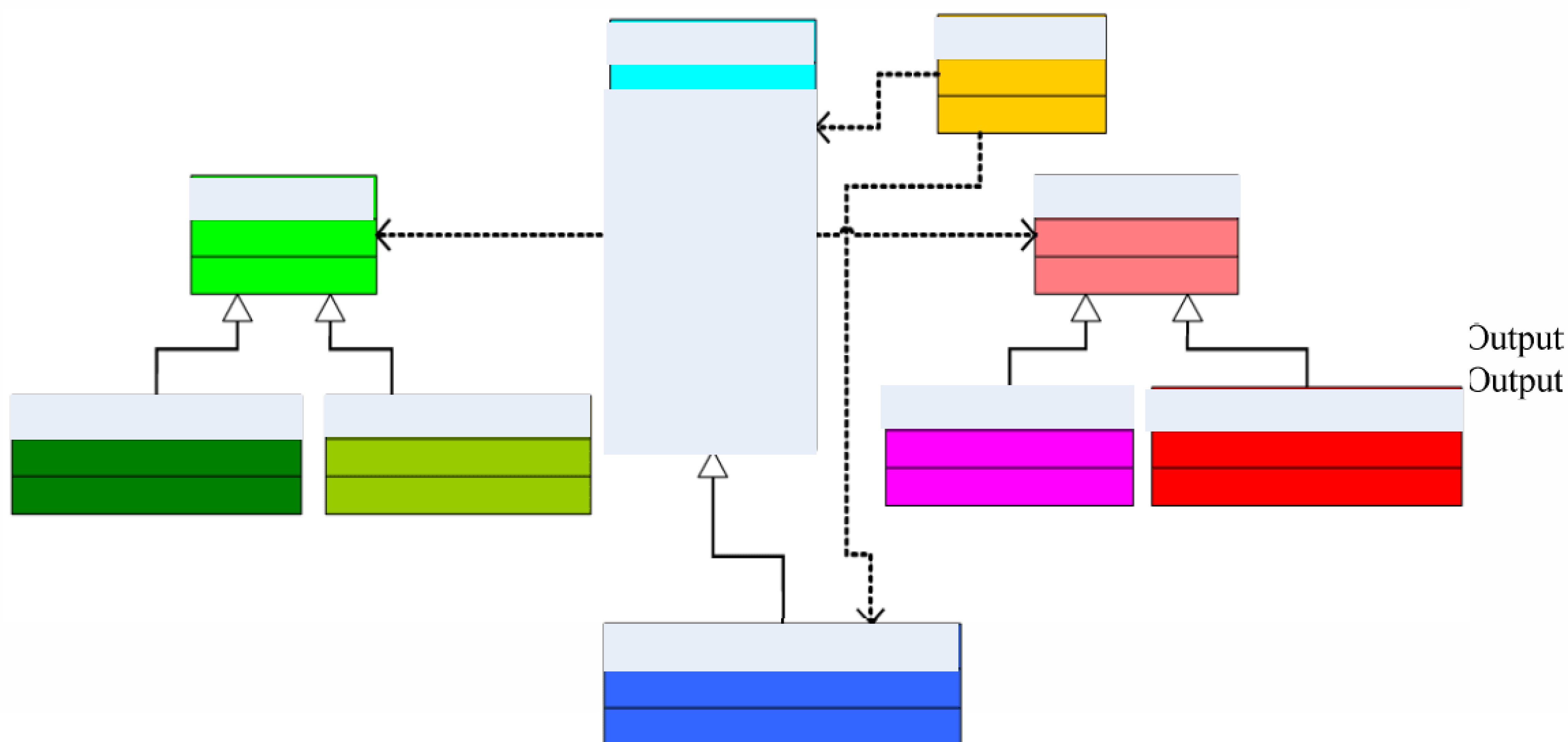


图 11-11　Player Engine的类结构

OpenCore 中的 Player Engine 具有清晰明确的接口，在此接口中，不同的系统可以根据具体情况实现不同的 Player。

Player Engine 位于 OpenCore 中的 engines/player/目录下，其中在 engines/player/include 目录中保存的是接口头文件，在 engines/player/src 目录中保存的是源文件和私有头文件。

(2) PVPlayer

PVPlayer 的结构如图 11-12 所示。

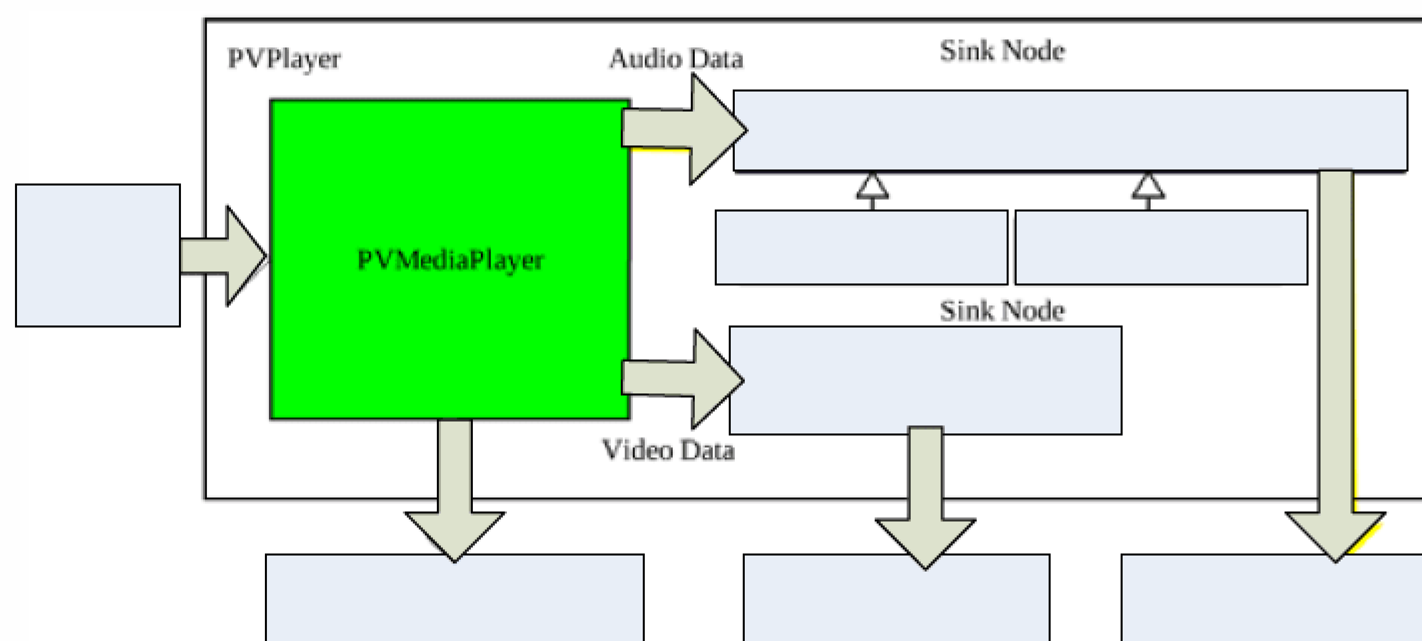


图 11-12　PVPlayer的结构

在图 11-12 中，Sink Node 会接受上一个 Node 写的动作。

PVPlayer 的类结构如图 11-13 所示。

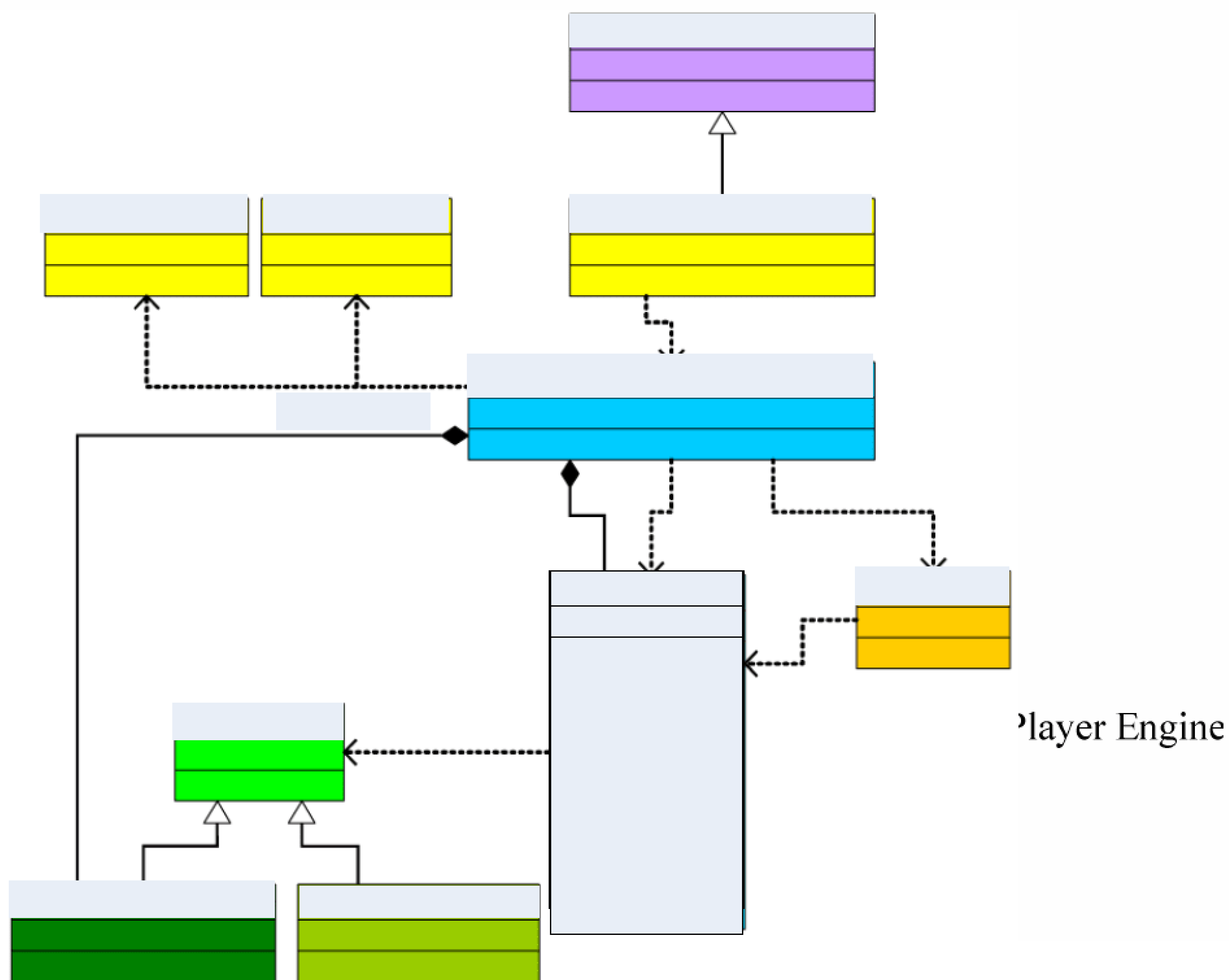


图 11-13　PVPlayer的类结构

(3) Author

OpenCore 的 Author 的编译文件是 pvauthor/Android.mk，编译后，将会生成动态库文件 libopencoreauthor.so，这个库与 Player 类似，在里面包含了如下两方面的内容：

- Author 的 Engine。
- 为 Android 构建的 Author。

OpenCore Author 的基本结构如图 11-14 所示。

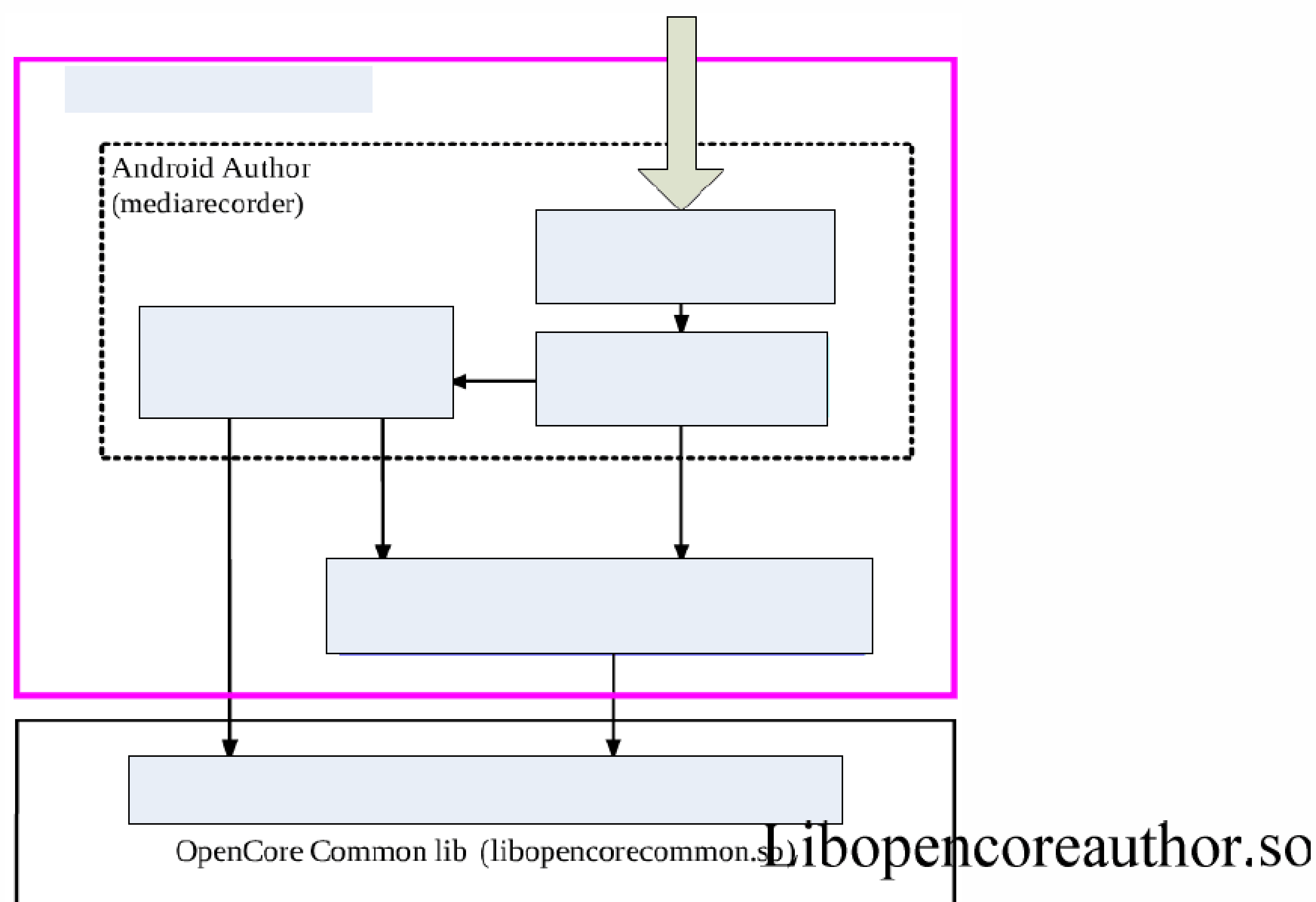


图 11-14　OpenCore Author的基本结构

在图 11-14 表示的结构中，目录 OpenCore/engines/author/是 Author 引擎目录，在里面主要包含了 include 和 src 两个子目录，其中如下两个头文件是接口：

```
pvauthorenginefactory.h
pvauthorengineinterface.h
```

OpenCore 的 Author 主要功能文件是 pvauthorengine.cpp，类 Author Engine 的结构如图 11-15 所示。

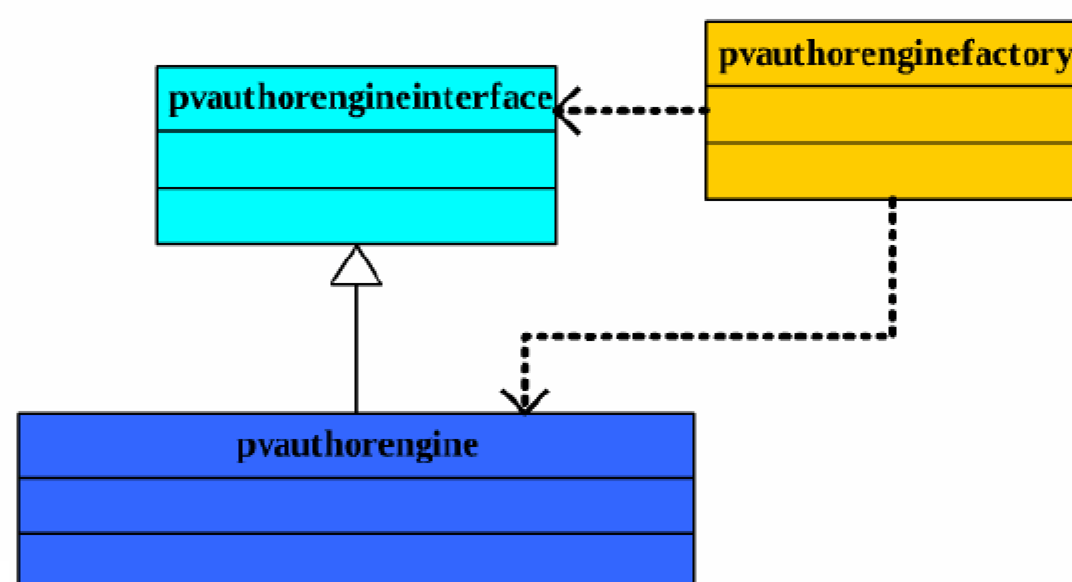


图 11-15　类Author Engine的结构

PV

PVAuthor 的结构如图 11-16 所示。

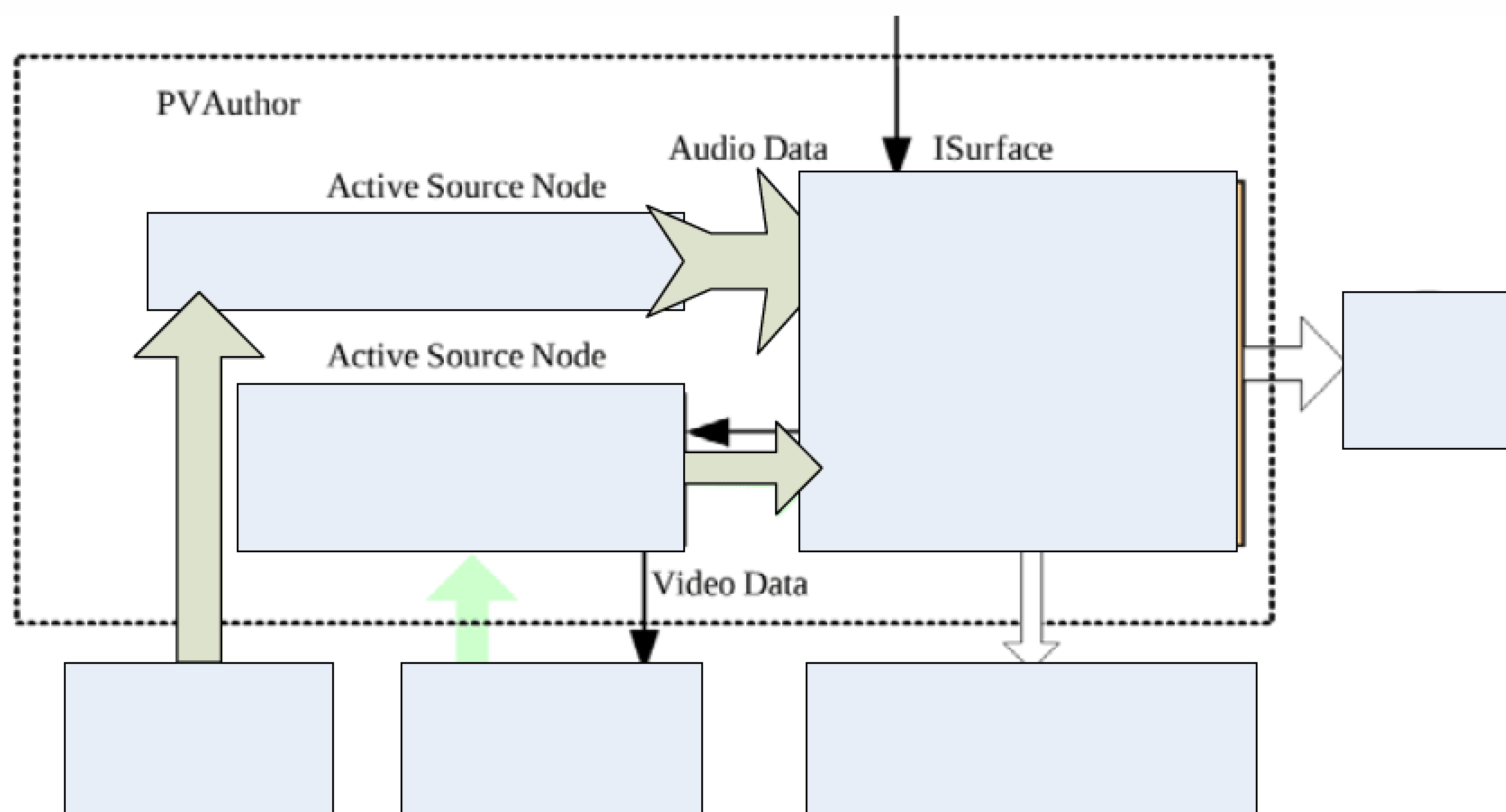


图 11-16　PVAuthor的结构

11.4　Stagefright框架详解

在 Android 系统中，预设的多媒体框架(Multimedia Framework)是 OpenCore。OpenCore 的特点是兼顾了跨平台的移植性，而且经过了多方验证，所以相对来说比较稳定；但是，其缺点是庞大复杂，需要耗费相当多的时间去维护。从 Android 2.0 开始，Google 引入了架构稍微简单的 Stagefright，并且有逐渐取代 OpenCore 的趋势。并且从 Android 2.2 开始，几乎完全放弃了 OpenCore，而主推 Stagefright。在本节的内容中，将详细讲解 Stagefright 框架的基本知识，为读者步入本书后面知识的学习打下基础。

11.4.1　Stagefright代码结构

Stagefright 是轻量级的多媒体框架，其主要功能是基于 OpenMAX 实现的。在 Stagefright 中提供了媒体播放等接口，这些接口可以为 Android 框架层所使用。

在 Android 开源代码中，Stagefright 的头文件路径如下所示：

```
frameworks/av/include/media/stagefright/
```

实现 Stagefright 功能的文件路径如下所示：

```
frameworks/av/media/libstagefright/
```

实现 Stagefright 播放器和录音器功能的文件路径如下所示：

```
frameworks/av/media/libmediaplayerservice/
```

测试 Stagefright 功能的代码路径如下所示：

```
frameworks/av/cmds/stagefright/
```

11.4.2 Stagefright实现OpenMAX接口

Android 系统中，Stagefright 可以实现 OpenMAX 接口，可以让 Stagefright 引擎内的 OMXCode 调用实现的 OpenMAX 接口，目的是使用 OpenMAX IL 实现“编码/解码”功能。

在 Android 系统中，通过 Stagefright 来定义 OpenMAX 接口，具体实现内容保存在 omx 目录中。在头文件 frameworks/av/media/libstagefright/include/OMX.h 中实现了 Android 标准的 IOMX 类，此文件的主要代码如下所示：

```
class OMX : public BnOMX, public IBinder::DeathRecipient {
public:
    OMX();
    virtual bool livesLocally(pid_t pid);
    virtual status_t listNodes(List<ComponentInfo> *list);
    virtual status_t allocateNode(
            const char *name, const sp<IOMXObserver> &observer, node_id *node);
    virtual status_t freeNode(node_id node);
    virtual status_t sendCommand(
            node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param);
    virtual status_t getParameter(
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size);
//……………………………………………………
    virtual status_t emptyBuffer(
            node_id node,
            buffer_id buffer,
            OMX_U32 range_offset, OMX_U32 range_length,
            OMX_U32 flags, OMX_TICKS timestamp);
    virtual status_t getExtensionIndex(
            node_id node,
            const char *parameter_name,
            OMX_INDEXTYPE *index);
    virtual sp<IOMXRenderer> createRenderer(
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight,
            int32_t rotationDegrees);
```

文件 frameworks/av/media/libstagefright/omx/OMX.cpp 是上述 OMX.h 的实现文件，首先定义函数 createRenderer()来创建映射，首先建立了一个 hardware renderer——SharedVideoRenderer (libstagefrighthw.so)，如果失败，则建立 software renderer——SoftwareRenderer(surface)。此函数的主要代码如下所示：

```
sp<IOMXRenderer> OMX::createRenderer(
        const sp<ISurface> &surface,
        const char *componentName,
```

```
    OMX_COLOR_FORMATTYPE colorFormat,
    size t encodedWidth, size t encodedHeight,
    size_t displayWidth, size_t displayHeight,
    int32_t rotationDegrees) {
Mutex::Autolock autoLock(mLock);
VideoRenderer *impl = NULL;
void *libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
if (libHandle) {
    typedef VideoRenderer *(*CreateRendererWithRotationFunc)(
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t displayWidth, size_t displayHeight,
            size_t decodedWidth, size_t decodedHeight,
            int32_t rotationDegrees);
    typedef VideoRenderer *(*CreateRendererFunc)(
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t displayWidth, size_t displayHeight,
            size t decodedWidth, size t decodedHeight);
    CreateRendererWithRotationFunc funcWithRotation =
        (CreateRendererWithRotationFunc)dlsym(
                libHandle,
                "_Z26createRendererWithRotationRKN7android2spINS_8",
                "ISurfaceEEEPKc20OMX_COLOR_FORMATTYPEjjjji");
    if (funcWithRotation) {
        impl = (*funcWithRotation)(
                surface, componentName, colorFormat,
                displayWidth, displayHeight, encodedWidth, encodedHeight,
                rotationDegrees);
    } else {
        CreateRendererFunc func =
            (CreateRendererFunc)dlsym(
                    libHandle,
                    "_Z14createRendererRKN7android2spINS_8ISurfaceEEEPKc20",
                    "OMX_COLOR_FORMATTYPEjjjj");
        if (func) {
            impl = (*func)(surface, componentName, colorFormat,
                    displayWidth, displayHeight, encodedWidth, encodedHeight);
        }
    }
    if (impl) {
        impl = new SharedVideoRenderer(libHandle, impl);
        libHandle = NULL;
    }
    if (libHandle) {
        dlclose(libHandle);
        libHandle = NULL;
```

```
        }
    }
    if (!impl) {
        LOGW("Using software renderer.");
        impl = new SoftwareRenderer(
                colorFormat,
                surface,
                displayWidth, displayHeight,
                encodedWidth, encodedHeight);
        if (((SoftwareRenderer*)impl)->initCheck() != OK) {
            delete impl;
            impl = NULL;
            return NULL;
        }
    }
    return new OMXRenderer(impl);
}
```

由此可见，OMXMaster 是 OMX.cpp 的真正实现者，并且能够管理 OpenMAX 插件的类，这些功能是通过头文件 OMXMaster.h 和源码文件 OMXMaster.cpp 实现的。

在文件 frameworks/av/include/media/stagefright/OMXMaster.h 中定义了类 OMXMaster，主要代码如下所示：

```
struct OMXCodec : public MediaSource, public MediaBufferObserver {
    enum CreationFlags {
        kPreferSoftwareCodecs    = 1,
        kIgnoreCodecSpecificData = 2,
        kClientNeedsFramebuffer  = 4,
    };
    static sp<MediaSource> Create(     //创建类MediaSource
            const sp<IOMX> &omx,
            const sp<MetaData> &meta, bool createEncoder,
            const sp<MediaSource> &source,
            const char *matchComponentName = NULL,
            uint32_t flags = 0);
    static void setComponentRole(      //设置组件的职责
            const sp<IOMX> &omx, IOMX::node_id node, bool isEncoder,
            const char *mime);
    virtual status_t start(MetaData *params = NULL);
    virtual status_t stop();
    virtual sp<MetaData> getFormat();
//省略声明函数代码
//……………………………………………………
```

在文件 frameworks/av/media/libstagefright/OMXMaster.cpp 中，定义静态函数 Create，将 MediaSource 作为 IOMX 插件给 OMXCode。函数 Create 的主要实现代码如下所示：

```
sp<MediaSource> OMXCodec::Create(
        const sp<IOMX> &omx,
```

```
    const sp<MetaData> &meta, bool createEncoder,
    const sp<MediaSource> &source,
    const char *matchComponentName,
    uint32_t flags) {
const char *mime;
bool success = meta->findCString(kKeyMIMEType, &mime); //获取 mime 信息
CHECK(success);
Vector<String8> matchingCodecs;
findMatchingCodecs(
        mime, createEncoder, matchComponentName, flags, &matchingCodecs);
if (matchingCodecs.isEmpty()) {
    return NULL;
}
sp<OMXCodecObserver> observer = new OMXCodecObserver;
IOMX::node_id node = 0;
const char *componentName;
for (size_t i=0; i<matchingCodecs.size(); ++i) { //使用 for 循环查找插件
    componentName = matchingCodecs[i].string();
    sp<MediaSource> softwareCodec = createEncoder?
        InstantiateSoftwareEncoder(componentName, source, meta)
        : InstantiateSoftwareCodec(componentName, source);
    if (softwareCodec != NULL) {
        LOGV("Successfully allocated software codec '%s'", componentName);
        return softwareCodec;
    }
    LOGV("Attempting to allocate OMX node '%s'", componentName);
    uint32_t quirks = getComponentQuirks(componentName, createEncoder);
    if (!createEncoder
      && (quirks & kOutputBuffersAreUnreadable)
      && (flags & kClientNeedsFramebuffer)) {
        if (strncmp(componentName, "OMX.SEC.", 8)) {
            LOGW("Component '%s' does not give the client access to "
                 "the framebuffer contents. Skipping.",
                 componentName);
            continue;
        }
    }
    status_t err = omx->allocateNode(componentName, observer, &node);
    if (err == OK) {
        LOGV("Successfully allocated OMX node '%s'", componentName);
        sp<OMXCodec> codec = new OMXCodec(          //新建类 OMXCodec
                omx, node, quirks,
                createEncoder, mime, componentName,
                source);
        observer->setCodec(codec);                  //设置编码/解码器
        err = codec->configureCodec(meta, flags);
        if (err == OK) {
            return codec;
        }
```

```
            LOGV("Failed to configure codec '%s'", componentName);
        }
    }
    return NULL;
}
```

11.4.3 分析Video Buffer的传输流程

视频播放的过程是处理 Video Buffer 的过程，在 Stagefright 框架中需要使用 VideoRenderer 插件来实现处理功能。接下来的内容中，将详细讲解在 Stagefright 框架中使用插件来传输 Video Buffer 的具体流程。

(1) OMXCodec 会在一开始的时候通过函数 read 来传送未解码的 data 数据给 decoder，并要求 decoder 将解码后的 data 传回来。对应的实现代码如下所示：

```
status_t OMXCodec::read(...)
{
  if (mInitialBufferSubmit)
  {
    mInitialBufferSubmit = false;
    drainInputBuffers(); <----- OMX_EmptyThisBuffer
    fillOutputBuffers(); <----- OMX_FillThisBuffer
  }
  ...
}
void OMXCodec::drainInputBuffers()
{
  Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];

  for (i=0; i<buffers->size(); ++i)
  {
    drainInputBuffer(&buffers->editItemAt(i));
  }
}
void OMXCodec::drainInputBuffer(BufferInfo *info)
{
  mOMX->emptyBuffer(...);
}
void OMXCodec::fillOutputBuffers()
{
  Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexOutput];
  for (i=0; i<buffers->size(); ++i)
  {
    fillOutputBuffer(&buffers->editItemAt(i));
  }
}
void OMXCodec::fillOutputBuffer(BufferInfo *info)
{
```

```
  mOMX->fillBuffer(...);
}
```

(2)　Decoder 从 input port(输入端)获取资料，然后进行解码处理，并回传 EmptyBufferDone 以通知 OMXCodec 当前的工作。对应的实现代码如下所示：

```
void OMXCodec::on_message(const omx_message &msg)
{
  switch (msg.type)
  {
    case omx_message::EMPTY_BUFFER_DONE:
    {
      IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
      drainInputBuffer(&buffers->editItemAt(i));
    }
  }
}
```

(3)　当 OMXCodec 接收到 EMPTY_BUFFER_DONE 之后，继续传送下一个未解码的资料给 Decoder。Decoder 解码后的资料送到 output port(输出端)，并回传 FillBufferDone 以通知 OMXCodec。对应的实现代码如下所示：

```
void OMXCodec::on_message(const omx_message &msg)
{
  switch (msg.type)
  {
    case omx message::FILL BUFFER DONE:
    {
      IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
      fillOutputBuffer(info);
      mFilledBuffers.push_back(i);
      mBufferFilled.signal();
    }
  }
}
```

当 OMXCodec 收到 FILL_BUFFER_DONE 后，将解码后的资料放入 mFilledBuffers，然后发出 mBufferFilled 信号，并要求 decoder 继续发出资料。

(4)　使用函数 read 等待 mBufferFilled 信号，当 mFilledBuffers 被填入资料后，函数 read 将其指定给 buffer，并回传给 AwesomePlayer。对应的实现代码如下所示：

```
status_t OMXCodec::read(MediaBuffer **buffer, ...)
{
  ...
  while (mFilledBuffers.empty())
  {
    mBufferFilled.wait(mLock);
  }
  BufferInfo *info = &mPortBuffers[kPortIndexOutput].editItemAt(index);
```

```
  info->mMediaBuffer->add_ref();
  *buffer = info->mMediaBuffer;
}
```

函数 AwesomePlayer::onVideoEvent 除了通过 OMXCodec::read 取得解码后的资料外，还需要将这些资料(mVideoBuffer)传给视频渲染器以便在屏幕上显示出来。此功能的实现过程如下所示。

(1) 在将 mVideoBuffer 中的资料输出之前，必须先建立 mVideoRenderer。对应的实现代码如下所示：

```
void AwesomePlayer::onVideoEvent()
{
  ...
  if (mVideoRenderer == NULL)
  {
    initRenderer_l();
  }
  ...
}
void AwesomePlayer::initRenderer_l()
{
  if (!strncmp("OMX.", component, 4))
  {
    mVideoRenderer = new AwesomeRemoteRenderer(
                        mClient.interface()->createRenderer(
                                     mISurface,
                                     component,
                                     ...));
  }
  else
  {
    mVideoRenderer = new AwesomeLocalRenderer(
                        ...,
                        component,
                        mISurface);
  }
}
```

(2) 如果视频渲染器是 OMX 组件，则需要建立一个 AwesomeRemoteRenderer 作为 mVideoRenderer。从上面步骤 1 中的代码看，AwesomeRemoteRenderer 的核心功能是由函数 OMX::createRenderer 实现的。createRenderer()先建立一个“硬件渲染器 SharedVideoRenderer (libstagefrighthw.so)”流程，如果失败，则建立“软件渲染器 SoftwareRenderer(surface)”流程。对应的实现代码如下所示：

```
sp<IOMXRenderer> OMX::createRenderer(...)
{
  VideoRenderer *impl = NULL;
  libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
```

```
  if (libHandle)
  {
    CreateRendererFunc func = dlsym(libHandle, ...);
    impl = (*func)(...); <----------------- Hardware Renderer
  }
  if (!impl)
  {
    impl = new SoftwareRenderer(...); <---- Software Renderer
  }
}
```

(3) 若视频解码器是软件组件，则需要建立 AwesomeLocalRenderer 作为 mVideoRenderer。AwesomeLocalRenderer 的构造函数会调用函数 init，其具体功能与函数 OMX::createRenderer 的相同。对应的实现代码如下所示：

```
void AwesomeLocalRenderer::init(...)
{
  mLibHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
  if (mLibHandle)
  {
    CreateRendererFunc func = dlsym(...);
    mTarget = (*func)(...); <---------------- Hardware Renderer
  }
  if (mTarget == NULL)
  {
    mTarget = new SoftwareRenderer(...); <--- Software Renderer
  }
}
```

(4) 建立 mVideoRenderer 后，就可以开始将解码后的资料回传给它，对应的实现代码如下所示：

```
void AwesomePlayer::onVideoEvent()
{
  if (!mVideoBuffer)
  {
    mVideoSource->read(&mVideoBuffer, ...);
  }
  [Check Timestamp]
  if (mVideoRenderer == NULL)
  {
    initRenderer_l();
  }
  mVideoRenderer->render(mVideoBuffer); <----- Render Data
}
```

经过上述操作之后，Renderer 的处理过程介绍完毕。在播放多媒体的时候，需要使用 audio 来实现处理功能。在 Stagefright 框架中，audio 的部分内容是由 AudioPlayer 来处理的，在函数 AwesomePlayer::play_l 中被建立。在接下来的内容中，介绍使用 audio 的基本流程。

(1) 当要求要求播放影音时，会同时建立并启动 AudioPlayer。对应的实现代码如下所示：

```
status_t AwesomePlayer::play_l()
{
  ...
  mAudioPlayer = new AudioPlayer(mAudioSink, ...);
  mAudioPlayer->start(...);
  ...
}
```

(2) 在启动 AudioPlayer 的过程中，会先读取第一笔解码后的资料，并开启 Audio Output。对应的实现代码如下所示：

```
status t AudioPlayer::start(...)
{
  mSource->read(&mFirstBuffer);
  if (mAudioSink.get() != NULL)
  {
    mAudioSink->open(..., &AudioPlayer::AudioSinkCallback, ...);
    mAudioSink->start();
  }
  else
  {
    mAudioTrack = new AudioTrack(..., &AudioPlayer::AudioCallback, ...);
    mAudioTrack->start();
  }
}
```

在上述代码中，AudioPlayer 并没有将 mFirstBuffer 传给 Audio Output。

(3) 在开启 Audio Output 的同时，AudioPlayer 将启用函数 callback()，这样每当函数 callback 被调用时，AudioPlayer 会到音频解码器读取解码后的资料。对应的实现代码如下所示：

```
size_t AudioPlayer::AudioSinkCallback(audioSink, buffer, size, ...)
{
  return fillBuffer(buffer, size);
}
void AudioPlayer::AudioCallback(..., info)
{
  buffer = info;
  fillBuffer(buffer->raw, buffer->size);
}
size_t AudioPlayer::fillBuffer(data, size)
{
  mSource->read(&mInputBuffer, ...);
  memcpy(data, mInputBuffer->data(), ...);
}
```

由上述代码可以知道，读取解码后 Audio 资料的工作是由函数 callback 驱动的，fillBuffer 会将资料(mInputBuffer)复制到数据 data 后，由 Audio Output 取用 data。

第 12 章 电源管理系统详解

在 Android 系统中，电源管理模块是 Android Power Management 系统，这是一个基于标准 Linux 电源管理的轻量级 Android 电源管理系统。在本章的内容中，将与读者一起探讨 Android 4.3 系统中 Power Management(电源管理)模块的基本知识，分析其具体原理和实现源码，为读者步入本书后面知识的学习打下基础。

12.1 Android Power Management基础

在 Android 系统中，电源管理模块(Android Power Management)的整体架构如图 12-1 所示。

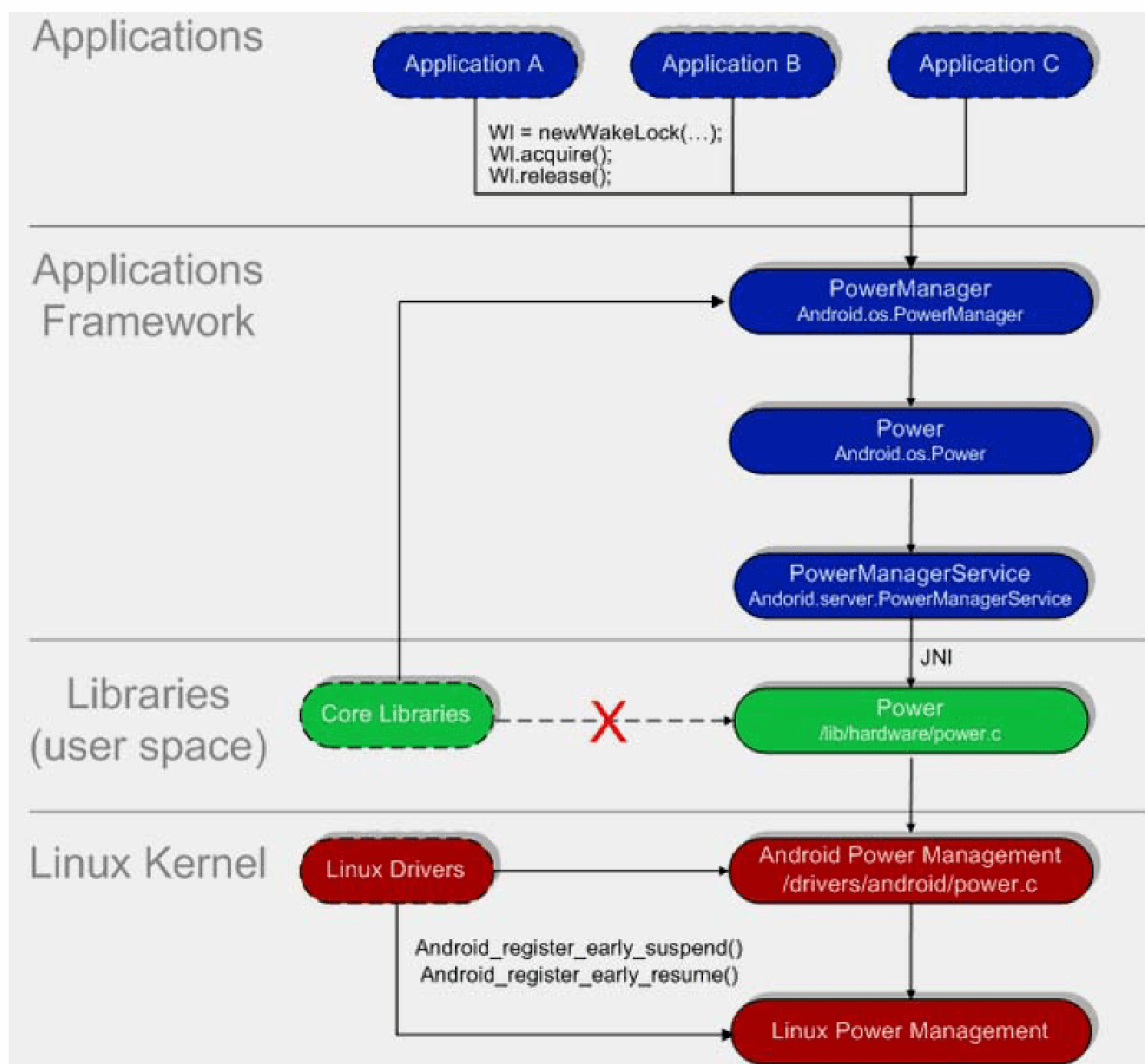


图 12-1　Android电源管理模块的整体架构

从图 12-1 给出的架构可以看出，电源管理主要是通过锁和定时器来切换系统的状态，使系统的功耗降至最低。为了使读者了解得更清晰，再看图 12-2 给出的电源管理详细架构。

由此可见，整个电源管理模块分为四大部分，分别是应用层、框架层、HAL 层和 Kernel 层。整个运作流程如下所示：

```
wake_lock → setScreenState(off) → request_suspend_state → early_suspend
 → wake_unlock → suspend → late suspend → sleep → wakeup → early resume
 → resume → late resume
```

在本章后面的内容中，将详细分析上述 4 个层次的具体实现过程。

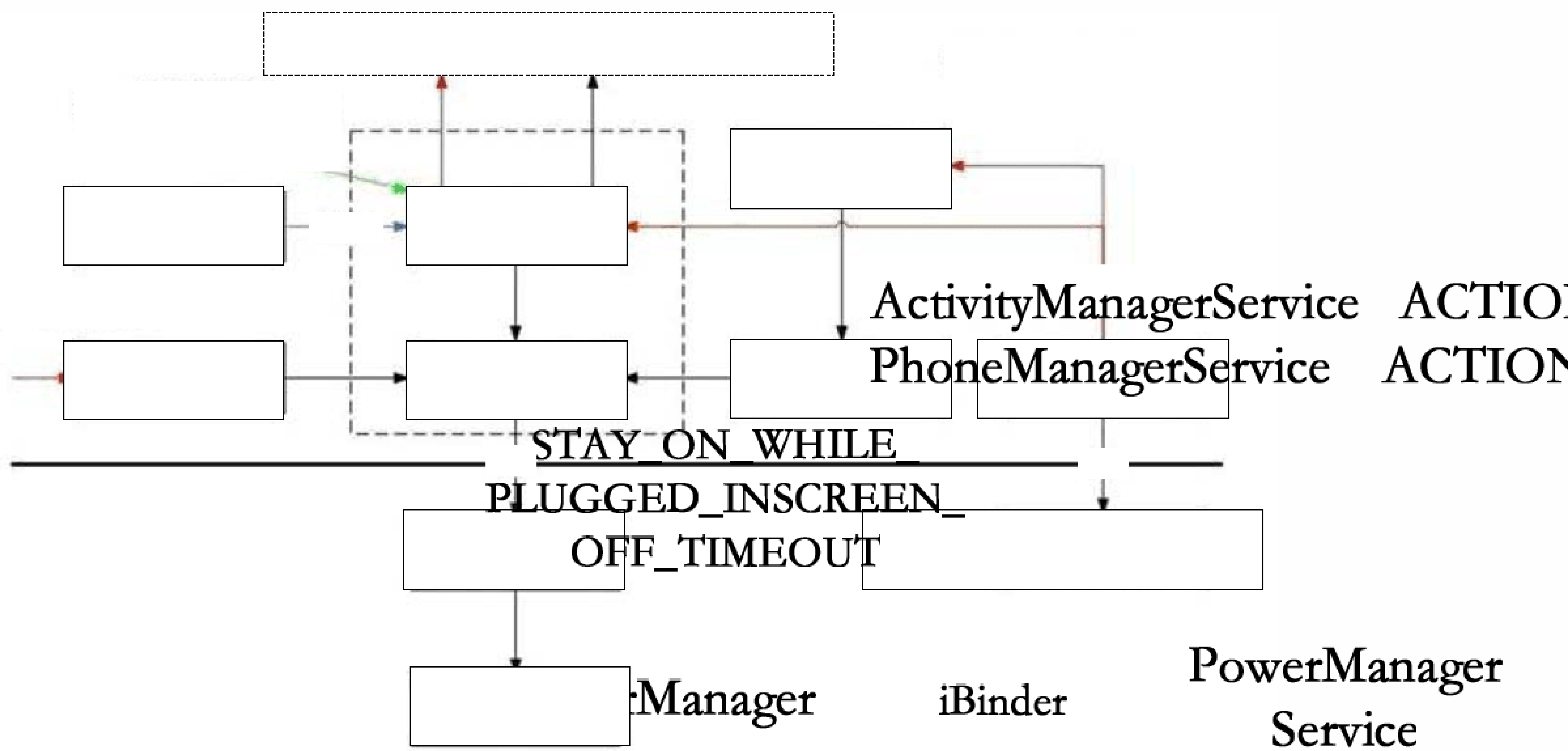


图 12-2　电源管理系统的详细架构

12.2　分析Framework层

我们知道，Android 系统的 Framework 层是接口层，为上面的应用层提供了拿来即用的接口。在 Android Power Management 系统中，Framework 层涉及了如下所示的文件：

- frameworks/base/core/java/android/os/PowerManager.java
- frameworks/base/services/java/com/android/server/power/PowerManagerService.java

在本节的内容中，我们将详细介绍上述 Power Management 系统中 Framework 层文件的实现过程。

12.2.1　文件PowerManager.java

文件 PowerManager.java 是提供给应用层调用的，核心是在文件 PowerManagerService.java 中实现的。在文件 PowerManager.java 中，定义了类 android.os.PowerManager，功能是控制设备电源状态的切换。PowerManager 在 getSystemService(Context.POWER_SERVICE)获取对象的时候是通过构造函数 PowerManager(IPowerManagerservice, Handler handler){}来创建的，而此处 IPowerManager 则是创建 PowerManager 实例的核心，IPowerManager 由 PowerManagerService 实现，所以从本质上说，PowerManager 的大部分方法是由 PowerManagerService 实现的。

在接下来的内容中，将详细分析文件 PowerManager.java 的具体实现流程。

(1)　定义类 PowerManager 和接口函数，主体代码如下所示：

```
public final class PowerManager {
    private static final String TAG = "PowerManager";
    public static final int PARTIAL_WAKE_LOCK = 0x00000001;
```

```
@Deprecated
public static final int SCREEN_DIM_WAKE_LOCK = 0x00000006;
@Deprecated
public static final int SCREEN_BRIGHT_WAKE_LOCK = 0x0000000a;
@Deprecated
public static final int FULL_WAKE_LOCK = 0x0000001a;
public static final int PROXIMITY_SCREEN_OFF_WAKE_LOCK = 0x00000020;
public static final int WAKE_LOCK_LEVEL_MASK = 0x0000ffff;
public static final int ACQUIRE_CAUSES_WAKEUP = 0x10000000;
public static final int ON_AFTER_RELEASE = 0x20000000;
public static final int WAIT_FOR_PROXIMITY_NEGATIVE = 1;
public static final int BRIGHTNESS_ON = 255;
```

(2) 定义对外接口函数，以实现对电源状态的控制管理。其中函数 goToSleep 的功能是强制设备进入 Sleep 状态，函数 wakeUp 的功能是强制设备进入 wakeUp 状态：

```
public void goToSleep(long time) {
    try {
        mService.goToSleep(time, GO_TO_SLEEP_REASON_USER);
    } catch (RemoteException e) {}
}
public void wakeUp(long time) {
    try {
        mService.wakeUp(time);
    } catch (RemoteException e) {}
}
```

(3) 定义函数 userActivity，功能是当发生 User Activity 事件时，电源设备会被切换到 Full on 的状态，并同时重置 Screen off 定时器，具体代码如下所示：

```
public void userActivity(long when, boolean noChangeLights) {
    try {
        mService.userActivity(when, USER_ACTIVITY_EVENT_OTHER,
                noChangeLights ? USER_ACTIVITY_FLAG_NO_CHANGE_LIGHTS : 0);
    } catch (RemoteException e) {}
}
```

12.2.2 文件PowerManagerService.java

文件 PowerManagerService.java 是 Power Management 系统中整个 Framework 层文件的核心，这个类的作用就是提供 PowerManager 的功能，以及负责整个电源管理状态机的运行。

PowerManagerService 服务是 Android 系统的上层电源管理服务，主要负责系统待机、屏幕背光、按键背光、键盘背光以及用户事件的处理。通过锁的申请与释放以及默认的待机时间来控制系统的待机状态；通过系统默认关闭屏的时间以及用户操作的事件状态控制背光的亮和暗。另外，PowerManagerService 服务还包括了光线、距离传感器上层查询与控制，LCD 亮度的调节最终也是由该服务完成的。在本章前面介绍的文件 PowerManager.java 只是定义了各个对外接口函数，而文件 PowerManagerService.java 则定义了各个接口函数的具体实现。

在接下来的内容中，将详细分析文件 PowerManagerService.java 的具体实现过程。

1. 定义常量和变量

在文件的开始定义服务类 PowerManagerService，并定义需要的变量和常量，主要实现代码如下所示：

```
private static final int LOCK_MASK = PowerManager.PARTIAL_WAKE_LOCK
                        | PowerManager.SCREEN_DIM_WAKE_LOCK
                        | PowerManager.SCREEN_BRIGHT_WAKE_LOCK
                        | PowerManager.FULL_WAKE_LOCK
                        | PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK;

//                 time since last state:           time since last event:
// The short keylight delay comes from secure settings; this is the default.
private static final int SHORT_KEYLIGHT_DELAY_DEFAULT = 6000; // t+6 sec
private static final int MEDIUM_KEYLIGHT_DELAY = 15000;      // t+15 sec
private static final int LONG_KEYLIGHT_DELAY = 6000;       // t+6 sec
private static final int LONG_DIM_TIME = 7000;           // t+N-5 sec

// How long to wait to debounce light sensor changes in milliseconds
private static final int LIGHT_SENSOR_DELAY = 2000; //光线传感器时延

// light sensor events rate in microseconds
private static final int LIGHT_SENSOR_RATE = 1000000; //光线传感器频率

// For debouncing the proximity sensor in milliseconds
private static final int PROXIMITY_SENSOR_DELAY = 1000; //距离传感器时延

// trigger proximity if distance is less than 5 cm
private static final float PROXIMITY_THRESHOLD = 5.0f; //距离传感器距离范围

// Cached secure settings; see updateSettingsValues()
private int mShortKeylightDelay = SHORT_KEYLIGHT_DELAY_DEFAULT;//键盘灯短暂时延

// Default timeout for screen off, if not found in settings database = 15 seconds.
private static final int DEFAULT_SCREEN_OFF_TIMEOUT = 15000;
  //默认屏幕超时时间，从 Settings 中获取

// flags for setPowerState
private static final int SCREEN_ON_BIT        = 0x00000001;
private static final int SCREEN_BRIGHT_BIT     = 0x00000002;
private static final int BUTTON_BRIGHT_BIT     = 0x00000004;
private static final int KEYBOARD_BRIGHT_BIT    = 0x00000008;
private static final int BATTERY_LOW_BIT      = 0x00000010;

// values for setPowerState

// SCREEN_OFF == everything off
```

```
private static final int SCREEN_OFF  = 0x00000000; //屏幕灭掉，进入睡眠状态

// SCREEN_DIM == screen on, screen backlight dim
private static final int SCREEN_DIM  = SCREEN_ON_BIT;//屏幕灭掉，依然在工作状态

// SCREEN_BRIGHT == screen on, screen backlight bright
private static final int SCREEN_BRIGHT =
  SCREEN_ON_BIT | SCREEN_BRIGHT_BIT; //屏幕亮，处于工作状态

// SCREEN_BUTTON_BRIGHT == screen on, screen and button backlights bright
private static final int SCREEN_BUTTON_BRIGHT =
  SCREEN_BRIGHT | BUTTON_BRIGHT_BIT; //屏幕亮，按键灯亮

// SCREEN_BUTTON_BRIGHT == screen on, screen, button and keyboard backlights bright
private static final int ALL_BRIGHT =
  SCREEN_BUTTON_BRIGHT | KEYBOARD_BRIGHT_BIT; //按键灯亮，键盘灯亮

// used for noChangeLights in setPowerState()
private static final int LIGHTS_MASK =
  SCREEN_BRIGHT_BIT | BUTTON_BRIGHT_BIT
   | KEYBOARD_BRIGHT_BIT; //屏幕亮，按键灯亮，键盘灯亮

boolean mAnimateScreenLights = true;

static final int ANIM_STEPS = 60/4;
// Slower animation for autobrightness changes
static final int AUTOBRIGHTNESS_ANIM_STEPS = 60;

// These magic numbers are the initial state of the LEDs at boot. Ideally
// we should read them from the driver, but our current hardware returns 0
// for the initial value.  Oops!
static final int INITIAL_SCREEN_BRIGHTNESS = 255;//屏幕初始状态 亮
static final int INITIAL_BUTTON_BRIGHTNESS =
  Power.BRIGHTNESS_OFF; //按键灯初始状态 灭
static final int INITIAL_KEYBOARD_BRIGHTNESS =
  Power.BRIGHTNESS_OFF; //键盘灯初始状态 灭

private final int MY_UID;
private final int MY_PID;

private boolean mDoneBooting = false;
private boolean mBootCompleted = false; //开机完成标志位
private int mStayOnConditions = 0;
private final int[] mBroadcastQueue = new int[] { -1, -1, -1 };
private final int[] mBroadcastWhy = new int[3];
private boolean mPreparingForScreenOn = false;
private boolean mSkippedScreenOn = false;
private boolean mInitialized = false;
```

```
private int mPartialCount = 0;
private int mPowerState;
// mScreenOffReason can be WindowManagerPolicy.OFF_BECAUSE_OF_USER,
// WindowManagerPolicy.OFF_BECAUSE_OF_TIMEOUT
// or WindowManagerPolicy.OFF_BECAUSE_OF_PROX_SENSOR
private int mScreenOffReason;
private int mUserState;
private boolean mKeyboardVisible = false;
private int mStartKeyThreshold = 0;
private boolean mUserActivityAllowed = true;
private int mProximityWakeLockCount = 0;
private boolean mProximitySensorEnabled = false; //距离传感器是否可用
private boolean mProximitySensorActive = false; //当前距离传感器是否工作
private int mProximityPendingValue = -1; // -1 == nothing, 0 == inactive, 1 == active
private long mLastProximityEventTime;
private int mScreenOffTimeoutSetting; //屏幕超时设置
private int mMaximumScreenOffTimeout = Integer.MAX_VALUE;
private int mKeylightDelay;
private int mDimDelay;
private int mScreenOffDelay;
private int mWakeLockState;
private long mLastEventTime = 0;
private long mScreenOffTime;
private volatile WindowManagerPolicy mPolicy;
private final LockList mLocks = new LockList();
private Intent mScreenOffIntent;
private Intent mScreenOnIntent;
private LightsService mLightsService; //系统 LightsService
private Context mContext;
private LightsService.Light mLcdLight; //屏
private LightsService.Light mButtonLight; //按键灯
private LightsService.Light mKeyboardLight; //键盘灯(若有实体输入法按键)
private LightsService.Light mAttentionLight; //通知等(若有信号灯)
private UnsynchronizedWakeLock mBroadcastWakeLock; //广播同步锁
private UnsynchronizedWakeLock mStayOnWhilePluggedInScreenDimLock;
private UnsynchronizedWakeLock mStayOnWhilePluggedInPartialLock;
private UnsynchronizedWakeLock mPreventScreenOnPartialLock;
private UnsynchronizedWakeLock mProximityPartialLock;
private HandlerThread mHandlerThread;
private HandlerThread mScreenOffThread;
private Handler mScreenOffHandler;
private Handler mHandler;

//计时器线程，主要完成管理屏幕超时操作，如当有用户点击屏幕时，
//该计时器重新开始计时，直到无任何操作，且到屏幕时延最大时间，将屏幕灭掉
private final TimeoutTask mTimeoutTask = new TimeoutTask();
private final BrightnessState mScreenBrightness =
  new BrightnessState(SCREEN_BRIGHT_BIT); //亮度管理
private boolean mStillNeedSleepNotification;
```

```
    private boolean mIsPowered = false;
    private IActivityManager mActivityService;
    private IBatteryStats mBatteryStats;
    private BatteryService mBatteryService; //电池服务
    private SensorManager mSensorManager; //Sensor 管理器
    private Sensor mProximitySensor; //距离传感器
    private Sensor mLightSensor; //光线传感器
    private Sensor mLightSensorKB; //光线传感器
    private boolean mLightSensorEnabled; //光线传感器是否可用
    private float mLightSensorValue = -1;
    private boolean mProxIgnoredBecauseScreenTurnedOff = false;
    private int mHighestLightSensorValue = -1;
    private boolean mLightSensorPendingDecrease = false;
    private boolean mLightSensorPendingIncrease = false;
    private float mLightSensorPendingValue = -1;
    private int mLightSensorScreenBrightness = -1;
    private int mLightSensorButtonBrightness = -1;
    private int mLightSensorKeyboardBrightness = -1;
    private boolean mDimScreen = true;
    private boolean mIsDocked = false;
    private long mNextTimeout;
    private volatile int mPokey = 0;
    private volatile boolean mPokeAwakeOnSet = false;
    private volatile boolean mInitComplete = false;
    private final HashMap<IBinder,PokeLock> mPokeLocks = new HashMap<IBinder,PokeLock>();
    // mLastScreenOnTime is the time the screen was last turned on
    private long mLastScreenOnTime;
    private boolean mPreventScreenOn;
    private int mScreenBrightnessOverride = -1;
    private int mButtonBrightnessOverride = -1;
    private int mScreenBrightnessDim;
    private boolean mUseSoftwareAutoBrightness;
    private boolean mAutoBrightessEnabled;
    private int[] mAutoBrightnessLevels;
    private int[] mLcdBacklightValues;
    private int[] mButtonBacklightValues;
    private int[] mKeyboardBacklightValues;
    private int mLightSensorWarmupTime;
    boolean mUnplugTurnsOnScreen;
    private int mWarningSpewThrottleCount;
    private long mWarningSpewThrottleTime;
    private int mAnimationSetting = ANIM_SETTING_OFF;

    // Must match with the ISurfaceComposer constants in C++.
    private static final int ANIM_SETTING_ON = 0x01;
    private static final int ANIM_SETTING_OFF = 0x10;

    // Used when logging number and duration of touch-down cycles
    private long mTotalTouchDownTime;
```

```
    private long mLastTouchDown;
    private int mTouchCycles;

    // could be either static or controllable at runtime
    private static final boolean mSpew = false;
    private static final boolean mDebugProximitySensor = (false || mSpew);
    private static final boolean mDebugLightSensor = (false || mSpew);

    private native void nativeInit();
    private native void nativeSetPowerState(
      boolean screenOn, boolean screenBright);
    private native void nativeStartSurfaceFlingerAnimation(int mode);
```

在文件 PowerManagerService 中，其中需要重点说明的变量如下所示。

- mDirty：功能是表示 power state(电源状态)的变化，在系统中一共定义了 12 个与之类似的变化，每一个 state 对应一个固定的数字，都是 2 的倍数。这样，当有若干个状态一起变化时，就会按位取或，这样不但会得到一个唯一的结果，而且可以准确地标示出各个状态的变化。
- mWakefulness：功能是表示 device 处于醒着的状态还是睡眠中的状态，或者处于两者之间的一种状态。这个状态与 display 的电源状态是不同的，display 的电源状态是独立管理的。这个变量用来表示 DIRTY_WAKEFULNESS 这个 power state 下的一个具体的内容。例如，当系统进入 Dreaming 的时候，首先变化的是 mDirty，在 mDirty 中对 DIRTY_WAKEFULNESS 置位，这说明系统中的 DIRTY_WAKEFULNESS 发生了变化。此时只是知道 DIRTY_WAKEFULNESS 发生了变化，并不知道 wakefulness 发生了怎样的变化。如果需要进一步了解系统的 wakefulness 变成了什么，则需要查看 mWakefulness 的内容。

2. 开机启动及处理

(1) 当 Android 系统启动时，会调用文件 SystemServer.java 中的接口函数 run，在此函数中，将 power 服务加入到系统服务中，具体代码如下所示：

```
power = new PowerManagerService();
ServiceManager.addService(Context.POWER_SERVICE, power);
```

其实在文件 SystemServer.java 中还定义了多种类型的服务，加入到了系统服务中，具体代码如下所示：

```
try {
    // Wait for installd to finished starting up so that it has a chance to
    // create critical directories such as /data/user with the appropriate
    // permissions.  We need this to complete before we initialize other services.
    Slog.i(TAG, "Waiting for installd to be ready.");
    installer = new Installer();
    installer.ping();
    Slog.i(TAG, "Power Manager");
    power = new PowerManagerService();
```

```
ServiceManager.addService(Context.POWER_SERVICE, power);
Slog.i(TAG, "Activity Manager");
context = ActivityManagerService.main(factoryTest);
Slog.i(TAG, "Display Manager");
display = new DisplayManagerService(context, wmHandler, uiHandler);
ServiceManager.addService(Context.DISPLAY_SERVICE, display, true);
Slog.i(TAG, "Telephony Registry");
telephonyRegistry = new TelephonyRegistry(context);
ServiceManager.addService("telephony.registry", telephonyRegistry);
Slog.i(TAG, "Scheduling Policy");
ServiceManager.addService(Context.SCHEDULING_POLICY_SERVICE,
  new SchedulingPolicyService());
AttributeCache.init(context);
if (!display.waitForDefaultDisplay()) {
    reportWtf("Timeout waiting for default display to be initialized.",
      new Throwable());
}
Slog.i(TAG, "Package Manager");
// Only run "core" apps if we're encrypting the device.
String cryptState = SystemProperties.get("vold.decrypt");
if (ENCRYPTING_STATE.equals(cryptState)) {
    Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
    onlyCore = true;
} else if (ENCRYPTED_STATE.equals(cryptState)) {
    Slog.w(TAG, "Device encrypted - only parsing core apps");
    onlyCore = true;
}
pm = PackageManagerService.main(context, installer,
  factoryTest!=SystemServer.FACTORY_TEST_OFF, onlyCore);
boolean firstBoot = false;
try {
    firstBoot = pm.isFirstBoot();
} catch (RemoteException e) {}
ActivityManagerService.setSystemProcess();
Slog.i(TAG, "Entropy Mixer");
ServiceManager.addService("entropy", new EntropyMixer(context));
Slog.i(TAG, "User Service");
ServiceManager.addService(Context.USER_SERVICE,
  UserManagerService.getInstance());
mContentResolver = context.getContentResolver();
// The AccountManager must come before the ContentService
try {
    Slog.i(TAG, "Account Manager");
    accountManager = new AccountManagerService(context);
    ServiceManager.addService(Context.ACCOUNT_SERVICE, accountManager);
} catch (Throwable e) {
    Slog.e(TAG, "Failure starting Account Manager", e);
}
Slog.i(TAG, "Content Manager");
```

```
    contentService = ContentService.main(context,
      factoryTest==SystemServer.FACTORY_TEST_LOW_LEVEL);
    Slog.i(TAG, "System Content Providers");
    ActivityManagerService.installSystemProviders();
    Slog.i(TAG, "Lights Service");
    lights = new LightsService(context);
    Slog.i(TAG, "Battery Service");
    battery = new BatteryService(context, lights);
    ServiceManager.addService("battery", battery);
    Slog.i(TAG, "Vibrator Service");
    vibrator = new VibratorService(context);
    ServiceManager.addService("vibrator", vibrator);
```

(2) 当光感服务与电池管理服务都开始启动后，开始进行初始化 power 服务的工作。初始化工作是通过调用函数 init 实现的，具体实现代码如下所示：

```
// only initialize the power service after we have started the
// lights service, content providers and the battery service.
power.init(context, lights, ActivityManagerService.self(), battery,
  BatteryStatsService.getService(), display);
Slog.i(TAG, "Alarm Manager");
alarm = new AlarmManagerService(context);
ServiceManager.addService(Context.ALARM_SERVICE, alarm);
Slog.i(TAG, "Init Watchdog");
Watchdog.getInstance().init(context, battery, power, alarm,
  ActivityManagerService.self());
```

在函数 init()中实现了一些基本的初始化工作，包括将 lights 和 battery 两个服务实例传入到 power 服务中，这两个服务将与 power 进行交互。除此之外，还开启了如下所示的两个线程。

① 开启处理亮度动画的线程函数 mScreenBrightnessAnimator.start()

mScreenBrightnessAnimator 是 PowerManagerService 子类 ScreenBrightnessAnimator 的实例，演示代码如下所示：

```
mHandlerThread = new HandlerThread("PowerManagerService");
mHandlerThread.start( );
```

② 初始化线程 initThread

当使用 start 启动电源管理服务后，会调用到 run 接口，并在其中回调到子类中的 protected void onLooperPrepared()，该接口又调用到 initInThread()，功能是实现一些值的初始化工作，并标识为“mInitComplete = true;”，发现为“true”的标识后，mHandlerThread.notifyAll()会通知创建 mHandlerThreadLooper 实例，该实例在函数 systemReady()中被 SystemSensorManager (mHandlerThread.getLooper())所使用。

另外，在 initThread 线程中还实现了对 mSettings.addObserver(settingsObserver)的监听，如果用户在系统中变更了关于背光时间或是否启用光感等服务的设置，PowerManagerService 可以获取到最新的状态值，这一功能需要使用函数 update()进行更新。

(3) 继续看函数 init()，最后调用函数 forceUserActivityLocked()来关闭服务，并设置标志

表示初始化工作完成。

3. 与系统其他模块之间的交互

在 Android 系统中，PowerManagerService 作为 Framework 中重要的能源管理模块，除了与应用程序交互之外，还需要与系统中的其他模块进行配合，在提供良好的能源管理同时，提供友好的用户体验。Android 系统除了提供公共接口与其他模块交互外，还提供了广播机制以对系统中发生的重要变化做出反应。表 12-1 中列出了在 PowerManagerService 中注册的 Receiver，以及这些 Receiver 监听的事件和处理方法。

表 12-1 PowerManagerService中注册的Receiver

Receiver 名称	相关变量	相关函数
BatteryReceiver	ACTION_BATTERY_CHANGED	handleBatterStateChangeLocked()
BootCompleteReceiver	ACTION_BOOT_COMPLETED	startWatchingForBootAnimationFinished()
userSwitchReceiver	ACTION_USER_SWITCHED	handleSettingsChangedLocked()
DockReceiver	ACTION_DOCK_EVENT	updatePowerStateLocked()
DreamReceiver	ACTION_DREAMING_STARTED ACTION_DREAMING_STOPPED	scheduleSandmanLocked()

文件 PowerManagerService.java 中除了这 5 个 Receiver 外，还定义了一个 SettingsObserver 以监视系统中以下属性的变化。

- SCREENSAVER_ENABLE：屏保的功能开启。
- SCREENSAVER_ACTIVE_ON_SLEEP：在睡眠时屏保启动。
- SCREENSAVER_ACTIVE_ON_DOCK：连接底座并且屏保启动。
- SCREEN_OFF_TIMEOUT：休眠时间。
- STAY_ON_PLUGGED_IN：有插入并且屏幕开启。
- SCREEN_BRIGHTNESS：屏幕的亮度。
- SCREEN_BRIGHTNESS_MODE：屏幕亮度的模式。

SettingObserver 会监视到上述属性发生的变化，并且会调用 SettingObserver 中的 onChange 方法：

```
public void onChange(boolean selfChange, Uri uri) {
    synchronized(mLock) {
        handleSettingsChangedLocked();
    }
}
```

由此可见，PowerManagerService 不但能够接收用户的请求，被动地去做一些操作，而且还要主动地监视系统中一些重要的属性变化和重要事件的发生。

4. 分析核心函数

(1) goToSleep

函数 goToSleep 的功能是进入休眠状态，具体实现代码如下所示：

```
@Override // Binder call
public void goToSleep(long eventTime, int reason) {
    if (eventTime > SystemClock.uptimeMillis()) {
        throw new IllegalArgumentException("event time must not be in the future");
    }
    //权限检查
    mContext.enforceCallingOrSelfPermission(
      android.Manifest.permission.DEVICE_POWER, null);

    final long ident = Binder.clearCallingIdentity();
    try {
        goToSleepInternal(eventTime, reason);//这里会调用函数的实现，
          //在 PowerManagerService 中有很多类似的使用方式，
          //之后的代码中会直接列出对应方法的实现
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
private void goToSleepInternal(long eventTime, int reason) {
    synchronized (mLock) {
        if (goToSleepNoUpdateLocked(eventTime, reason)) {
            updatePowerStateLocked();
        }
    }
}
```

(2) goToSleepNoUpdateLocked

函数 goToSleepNoUpdateLocked 是 goToSleep 功能的计算者，功能是确定是否要休眠，具体实现代码如下所示：

```
private boolean goToSleepNoUpdateLocked(long eventTime, int reason) {
    if (DEBUG_SPEW) {
        Slog.d(TAG, "goToSleepNoUpdateLocked: eventTime=" + eventTime + ",
          reason=" + reason);
    }
    if (eventTime<mLastWakeTime || mWakefulness==WAKEFULNESS_ASLEEP
      || !mBootCompleted || !mSystemReady) {
        return false;
    }
    switch (reason) {
        case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
            Slog.i(TAG, "Going to sleep due to device administration policy...");
            break;
        case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT:
            Slog.i(TAG, "Going to sleep due to screen timeout...");
            break;
        default:
            Slog.i(TAG, "Going to sleep by user request...");
            reason = PowerManager.GO_TO_SLEEP_REASON_USER;
```

```
                break;
        }
        sendPendingNotificationsLocked();
        mNotifier.onGoToSleepStarted(reason);
        mSendGoToSleepFinishedNotificationWhenReady = true;

        mLastSleepTime = eventTime;
        mDirty |= DIRTY_WAKEFULNESS;
        mWakefulness = WAKEFULNESS_ASLEEP;

        // Report the number of wake locks that will be cleared by going to sleep.
        int numWakeLocksCleared = 0;
        final int numWakeLocks = mWakeLocks.size();
        for (int i=0; i<numWakeLocks; i++) {
            final WakeLock wakeLock = mWakeLocks.get(i);
            switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
                case PowerManager.FULL_WAKE_LOCK:
                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
                case PowerManager.SCREEN_DIM_WAKE_LOCK:
                    numWakeLocksCleared += 1;
                    break;
            }
        }
        EventLog.writeEvent(
          EventLogTags.POWER_SLEEP_REQUESTED, numWakeLocksCleared);
        return true;
}
```

通过上述代码可知，在此并没有真正地让设备进入到 sleep 休眠状态，而仅仅是对 PowerManagerService 中一些必要的属性进行了赋值处理。

正因为如此，所以在 Android 系统中，可以把多个电源状态属性的多个变化放在一起共同执行，而真正的功能执行者就是 updatePowerStateLocked。

注意： 在 PowerManagerService 的具体实现代码中，有很多含有“xxxNoUpdateLocked”格式后缀的函数名，其实现原理都类似于函数 goToSleepNoUpdateLocked。

(3) updatePowerStateLocked

函数 updatePowerStateLocked 的功能是更新电源状态的锁定，也就是把影响到 Power Management 的变化放在一起进行更新，让电源管理机制能够真正地起到作用。

函数 updatePowerStateLocked 的具体实现代码如下所示：

```
private void updatePowerStateLocked() {
    if (!mSystemReady || mDirty == 0) {
    //如果系统没有准备好，或者 power state 没有发生任何变化，这个方法可以不用执行
        return;
    }
    // Phase 0: Basic state updates.
    updateIsPoweredLocked(mDirty);
```

```
    updateStayOnLocked(mDirty);
    // Phase 1: Update wakefulness.
    // Loop because the wake lock and user activity computations are influenced
    // by changes in wakefulness.
    final long now = SystemClock.uptimeMillis();
    int dirtyPhase2 = 0;
    for (;;) {
        int dirtyPhase1 = mDirty;
        dirtyPhase2 |= dirtyPhase1;
        mDirty = 0;
        updateWakeLockSummaryLocked(dirtyPhase1);
        //前面解释几个变量的时候，就已经提到了 WakeLockSummary 和 UserActivitySummary
        updateUserActivitySummaryLocked(now, dirtyPhase1);//在这里的两个方法中已经开
          //始用到了。想必通过方法名，大概也已经能了解其功能了
        if (!updateWakefulnessLocked(dirtyPhase1)) {
            break;
        }
    }
    // Phase 2: Update dreams and display power state.
    updateDreamLocked(dirtyPhase2);
    updateDisplayPowerStateLocked(dirtyPhase2);
    // Phase 3: Send notifications, if needed.
    if (mDisplayReady) {
        sendPendingNotificationsLocked();
    }
    // Phase 4: Update suspend blocker.
    // Because we might release the last suspend blocker here, we need to make sure
    // we finished everything else first!
    updateSuspendBlockerLocked();
}
```

通过上述实现代码可知，函数 updatePowerStateLocked 按照如下 4 个阶段对 Power State(电源状态)进行更新。

① 第 1 阶段：基本状态的更新。

首先执行函数 updateIsPoweredLocked，功能是判断设备是否处于充电状态中，如果 DIRTY_BATTERY_STATE 发生了变化，说明设备的电池的状态有过改变。

函数 updateIsPoweredLocked 的具体实现代码如下所示：

```
/**
* Updates the value of mIsPowered.
* Sets DIRTY_IS_POWERED if a change occurred.
*/
private void updateIsPoweredLocked(int dirty) {
    if ((dirty & DIRTY_BATTERY_STATE) != 0) {
        final boolean wasPowered = mIsPowered;
        final int oldPlugType = mPlugType;
        mIsPowered =
          mBatteryService.isPowered(BatteryManager.BATTERY_PLUGGED_ANY);
```

```
        mPlugType = mBatteryService.getPlugType();
        mBatteryLevel = mBatteryService.getBatteryLevel();

        if (DEBUG) {
            Slog.d(TAG, "updateIsPoweredLocked: wasPowered=" + wasPowered
                    + ", mIsPowered=" + mIsPowered
                    + ", oldPlugType=" + oldPlugType
                    + ", mPlugType=" + mPlugType
                    + ", mBatteryLevel=" + mBatteryLevel);
        }

        if (wasPowered!=mIsPowered || oldPlugType!=mPlugType) {
            mDirty |= DIRTY_IS_POWERED;

            // Update wireless dock detection state.
            final boolean dockedOnWirelessCharger =
              mWirelessChargerDetector.update(
                mIsPowered, mPlugType, mBatteryLevel);
            final long now = SystemClock.uptimeMillis();
            if (shouldWakeUpWhenPluggedOrUnpluggedLocked(
              wasPowered, oldPlugType, dockedOnWirelessCharger)) {
                wakeUpNoUpdateLocked(now);
            }
            userActivityNoUpdateLocked(now,
              PowerManager.USER_ACTIVITY_EVENT_OTHER, 0, Process.SYSTEM_UID);

            if (dockedOnWirelessCharger) {
                mNotifier.onWirelessChargingStarted();
            }
        }
    }
}
```

然后通过对比、判断(通过电池状态前后的变化和充电状态的变化来判断)确定是否处于充电状态，会在 mDirty 中标记出充电方式的改变，并同时根据充电状态的变化进行一些相应的处理。

接着执行函数 updateStayOnLocked，功能是更新 device 是否处于开启状态。

此函数也是通过 mStayOn 发生的前后变化作为判断依据的，如果 device 的属性 Settings.Global.STAY_ON_WHILE_PLUGGED_IN 为置位，并且没有达到电池充电时持续开屏时间的最大值(也就是说，在插入电源后的一段时间内保持开屏状态)，那么 mStayOn 为真。

函数 updateStayOnLocked 的具体实现代码如下所示：

```
private void updateStayOnLocked(int dirty) {
    if ((dirty & (DIRTY_BATTERY_STATE | DIRTY_SETTINGS)) != 0) {
        final boolean wasStayOn = mStayOn;
        if (mStayOnWhilePluggedInSetting!=0
          && !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked()) {
            mStayOn = mBatteryService.isPowered(mStayOnWhilePluggedInSetting);
```

```
        } else {
            mStayOn = false;
        }
        if (mStayOn != wasStayOn) {
            mDirty |= DIRTY_STAY_ON;
        }
    }
}
```

由此可见，在第一阶段的更新过程中，主要是进行了充电状态的判断工作，然后根据充电的状态，更新了一些必要的属性的变化，同时更新了 mDirty。

② 第 2 阶段：显示内容的更新。

mWakefuless 表示 device 处于的醒着或睡眠或两者之间的一种状态，这种状态会影响到 wake lock(唤醒锁)和 user activity(用户活动)的计算，所以要进行更新。在第 2 阶段先通过一个死循环进行处理，只有当 updateWakefulnessLocked 返回为 false 时，才能跳出这个循环。在刚刚进入这个循环的时候，对 mDirty 进行了重置，这能够说明在这次 updatePowerState 后会执行前面所有发生的电源状态，而不会让其影响到下一次的变化。同时也在为下一次的电源状态从头开始更新做好准备。

其中函数 updateWakeLockSummaryLocked 的实现代码如下所示：

```
private void updateWakeLockSummaryLocked(int dirty) {
    if ((dirty&(DIRTY_WAKE_LOCKS|DIRTY_WAKEFULNESS)) != 0) {
        mWakeLockSummary = 0;

        final int numWakeLocks = mWakeLocks.size();
        for (int i=0; i<numWakeLocks; i++) {
            final WakeLock wakeLock = mWakeLocks.get(i);
            switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
                case PowerManager.PARTIAL_WAKE_LOCK:
                    mWakeLockSummary |= WAKE_LOCK_CPU;
                    break;
                case PowerManager.FULL_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU
                          | WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_BUTTON_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |=WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
```

```
                case PowerManager.SCREEN_DIM_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_DIM;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |=
                         WAKE_LOCK_CPU | WAKE_LOCK_PROXIMITY_SCREEN_OFF;
                    }
                    break;
            }
        }
        if (DEBUG_SPEW) {
            Slog.d(TAG,
              "updateWakeLockSummaryLocked: mWakefulness="
                + wakefulnessToString(mWakefulness)
                + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary));
        }
    }
}
```

函数 updateUserActivitySummaryLocked 实现了锁屏时间和变暗时间的比较，假设在系统中设置的睡眠时间是 30s，而在 PowerManagerService 中默认的 SCREEN_DIM_DURATION 是 7s，则说明如果没有用户活动的话，设备屏幕在第 23s 开始变换，持续 7s 时间，然后才开始关闭屏幕。函数 updateUserActivitySummaryLocked 的具体实现代码如下所示：

```
private void updateUserActivitySummaryLocked(long now, int dirty) {
    // Update the status of the user activity timeout timer.
    if ((dirty&(DIRTY_USER_ACTIVITY|DIRTY_WAKEFULNESS|DIRTY_SETTINGS)) != 0) {
        mHandler.removeMessages(MSG_USER_ACTIVITY_TIMEOUT);
        long nextTimeout = 0;
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            final int screenOffTimeout = getScreenOffTimeoutLocked();
            final int screenDimDuration =
              getScreenDimDurationLocked(screenOffTimeout);
            mUserActivitySummary = 0;
            if (mLastUserActivityTime >= mLastWakeTime) {
                nextTimeout = mLastUserActivityTime
                  + screenOffTimeout - screenDimDuration;
                if (now < nextTimeout) {
                    mUserActivitySummary |= USER_ACTIVITY_SCREEN_BRIGHT;
                } else {
                    nextTimeout = mLastUserActivityTime + screenOffTimeout;
                    if (now < nextTimeout) {
                        mUserActivitySummary |= USER_ACTIVITY_SCREEN_DIM;
```

```
                    }
                }
            }
            if (mUserActivitySummary==0
              && mLastUserActivityTimeNoChangeLights>=mLastWakeTime) {
                nextTimeout =
                  mLastUserActivityTimeNoChangeLights + screenOffTimeout;
                if (now<nextTimeout
                  && mDisplayPowerRequest.screenState
                  !=DisplayPowerRequest.SCREEN_STATE_OFF) {
                    mUserActivitySummary = mDisplayPowerRequest.screenState
                      ==DisplayPowerRequest.SCREEN_STATE_BRIGHT ?
                      USER_ACTIVITY_SCREEN_BRIGHT : USER_ACTIVITY_SCREEN_DIM;
                }
            }
            if (mUserActivitySummary != 0) {
                Message msg = mHandler.obtainMessage(MSG_USER_ACTIVITY_TIMEOUT);
                msg.setAsynchronous(true);
                mHandler.sendMessageAtTime(msg, nextTimeout);
            }
        } else {
            mUserActivitySummary = 0;
        }
        if (DEBUG_SPEW) {
            Slog.d(TAG, "updateUserActivitySummaryLocked: mWakefulness="
                + wakefulnessToString(mWakefulness)
                + ", mUserActivitySummary=0x"
                + Integer.toHexString(mUserActivitySummary)
                + ", nextTimeout=" + TimeUtils.formatUptime(nextTimeout));
        }
    }
}
```

具体在什么时候才可以跳出这个循环，需要视函数 updateWakefulnessLocked 的返回值而定。函数 updateWakefulnessLocked 的具体实现代码如下所示：

```
/**
* 这个方法的功能是：根据当前的 wakeLocks 和用户的活动情况，来决定设备是否需要休眠
*
*///当 wakefulness 发生变化的时，返回 true，同时也需要重新计算 power state
private boolean updateWakefulnessLocked(int dirty) {
    boolean changed = false;
    if ((dirty&(DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
      | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
      | DIRTY_DOCK_STATE)) != 0) {
        if (mWakefulness==WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakefulnessLocked: Bed time...");
            }
```

```
            final long time = SystemClock.uptimeMillis();
            if (shouldNapAtBedTimeLocked()) {
                changed = napNoUpdateLocked(time);
            } else {
                changed = goToSleepNoUpdateLocked(time,
                  PowerManager.GO_TO_SLEEP_REASON_TIMEOUT);
            }
        }
    }
    return changed;
}
```

在上述代码中，用到了函数 isItBedTimeYetLocked，功能是询问是否到了应该休眠的时间了，如果现在设备处于醒着的状态，就马上改变为睡眠时间。函数 isItBedTimeYetLocked 的具体实现代码如下所示：

```
private boolean isItBedTimeYetLocked() {
    return mBootCompleted && !isBeingKeptAwakeLocked();
}
```

在上述代码中，如果有应用程序持有 wakelock 或产生了用户活动，或者处于充电状态，那么 isBeingKeptAwakeLocked 的返回值就是 true，相应地 isItBedTimeYetLocked 返回值为 false，因为还有没释放的 wakelock，或者有用户活动，或者是在充电等，这说明还没有到睡眠的时间。但是，如果 wakelock 都释放了，并且也没有用户活动了，那么就可以进入睡眠状态。这时的设备由醒着状态转换为睡眠的处理过程，此过程需要调用函数 updateWakefulnessLocked 来实现，此函数的具体实现代码如下所示：

```
private boolean shouldNapAtBedTimeLocked() {
    return mDreamsActivateOnSleepSetting
      || (mDreamsActivateOnDockSetting
      && mDockState!=Intent.EXTRA_DOCK_STATE_UNDOCKED);
}
```

代码中，mDreamsActivateOnSleepSetting 的默认值为 false，mDreamsActivateOnDockSetting 的默认值为 true。因为一般的用户没有接入 Dock，所以 mDockState!=Intent.EXTRA_DOCK_STATE_UNDOCKED 的值为 false，所以上述函数的返回值是 false。这样接下来需要执行的函数就是 goToSleepNoUpdateLocked，此函数已经在前面进行了讲解，它只是更新了 power state 中一些必要的属性，并没有真正执行能够让 Device 进入 sleep 的代码，真正执行的代码在函数 updatePowerStateLocked 中。

③ 第 3 阶段：dream 和 display 状态的更新。

在这一阶段，函数 updateDreamLocked 会根据 mDirty 的变化，并结合其他的属性共同判断是否要开始屏保(Dreaming)处理。

如果需要开始进行屏保处理的话，需要通过 DreamManagerService 开启 Dreaming。

函数 updateDreamLocked 的具体实现代码如下所示：

```
private void updateDreamLocked(int dirty) {
    if ((dirty&(DIRTY_WAKEFULNESS
```

```
      | DIRTY_USER_ACTIVITY
      | DIRTY_WAKE_LOCKS
      | DIRTY_BOOT_COMPLETED
      | DIRTY_SETTINGS
      | DIRTY_IS_POWERED
      | DIRTY_STAY_ON
      | DIRTY_PROXIMITY_POSITIVE
      | DIRTY_BATTERY_STATE)) != 0) {
        scheduleSandmanLocked();
    }
}
```

函数 updateDisplayPowerStateLocked 的主要功能是每次重新计算 Display Power State 的值，即 SCREEN_STATE_OFF、SCREEN_STATE_DIM 和 SCREEN_STATE_BRIGHT 这三个值之一。如果在 DisplayController 中更新了 Display Power State 的值，那么 DisplayController 会发送通知消息，所以还需要回来重新检查一次。函数 updateDisplayPowerStateLocked 的具体实现代码如下所示：

```
private void updateDisplayPowerStateLocked(int dirty) {
    if ((dirty&(DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS
      | DIRTY_ACTUAL_DISPLAY_POWER_STATE_UPDATED | DIRTY_BOOT_COMPLETED
      | DIRTY_SETTINGS | DIRTY_SCREEN_ON_BLOCKER_RELEASED)) != 0) {
        int newScreenState = getDesiredScreenPowerStateLocked();
          //获取 display 的 power state 将要变成的状态

        if (newScreenState != mDisplayPowerRequest.screenState)
        {//mDisplayPowerRequest.screenState 是目前 display 所处的 power state
            if (newScreenState == DisplayPowerRequest.SCREEN_STATE_OFF
              && mDisplayPowerRequest.screenState
              != DisplayPowerRequest.SCREEN_STATE_OFF)
            {//这个判断意味着：目前 display 的电源状态不是 OFF，但是想要变为 OFF
                mLastScreenOffEventElapsedRealTime =
                  SystemClock.elapsedRealtime();
            }
            mDisplayPowerRequest.screenState = newScreenState;
            nativeSetPowerState(
              newScreenState!=DisplayPowerRequest.SCREEN_STATE_OFF,
              newScreenState==DisplayPowerRequest.SCREEN_STATE_BRIGHT);
        }
        int screenBrightness = mScreenBrightnessSettingDefault;
        float screenAutoBrightnessAdjustment = 0.0f;
        boolean autoBrightness = (mScreenBrightnessModeSetting ==
          Settings.System.SCREEN_BRIGHTNESS_MODE_AUTOMATIC);
          //获取屏幕亮度模式是否为自动变化

        if (isValidBrightness(mScreenBrightnessOverrideFromWindowManager))
        {//mScreenBrightnessOverideFromeWindowManager 是 WindowManager 设置的亮度
         //大小，默认值为-1
            screenBrightness = mScreenBrightnessOverrideFromWindowManager;
```

```
            autoBrightness = false;
        } else if (isValidBrightness(mTemporaryScreenBrightnessSettingOverride))
        {//mTemporaryScreenBrightnessSettingOverride在widget中设置的临时亮度大小，
         //默认为-1
            screenBrightness = mTemporaryScreenBrightnessSettingOverride;
        } else if (isValidBrightness(mScreenBrightnessSetting))
        {//在Settings中设置的默认亮度，在Android 4.2中其值为102
            screenBrightness = mScreenBrightnessSetting;
        }
        if (autoBrightness) { //如果亮度是自动调节的话
            screenBrightness = mScreenBrightnessSettingDefault;
            if (isValidAutoBrightnessAdjustment(
              mTemporaryScreenAutoBrightnessAdjustmentSettingOverride)) {
                screenAutoBrightnessAdjustment =
                  mTemporaryScreenAutoBrightnessAdjustmentSettingOverride;
            } else if (isValidAutoBrightnessAdjustment(
              mScreenAutoBrightnessAdjustmentSetting)) {
                screenAutoBrightnessAdjustment =
                  mScreenAutoBrightnessAdjustmentSetting;
            }
        }
        screenBrightness = Math.max(Math.min(screenBrightness,
          mScreenBrightnessSettingMaximum), mScreenBrightnessSettingMinimum);
        screenAutoBrightnessAdjustment = Math.max(Math.min(
          screenAutoBrightnessAdjustment, 1.0f), -1.0f);
        mDisplayPowerRequest.screenBrightness = screenBrightness;//从这行向下开始
          //就是配置完成DisplayPowerRequest，然后以此为参数，
          //通过requestPowerSTate方法进行设置
        mDisplayPowerRequest.screenAutoBrightnessAdjustment =
          screenAutoBrightnessAdjustment;
        mDisplayPowerRequest.useAutoBrightness = autoBrightness;
        mDisplayPowerRequest.useProximitySensor =
          shouldUseProximitySensorLocked();
        mDisplayPowerRequest.blockScreenOn = mScreenOnBlocker.isHeld();
        mDisplayReady = mDisplayPowerController
          .requestPowerState(mDisplayPowerRequest,
          mRequestWaitForNegativeProximity);
        mRequestWaitForNegativeProximity = false;
        if (DEBUG_SPEW) {
            Slog.d(TAG, "updateScreenStateLocked: mDisplayReady=" + mDisplayReady
              + ", newScreenState=" + newScreenState
              + ", mWakefulness=" + mWakefulness
              + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary)
              + ", mUserActivitySummary=0x"
              + Integer.toHexString(mUserActivitySummary)
              + ", mBootCompleted=" + mBootCompleted);
        }
    }
}
```

在上述代码中，实现了对屏幕亮度的请求，并改变了屏幕的亮度。在调用的函数 goToSleepNoUpdateLocked 中，Display State 会因为 requestPowerState 函数的调用而起作用。

④　第 4 阶段：Suspend Blocker 的更新。

之所以放在最后一步才进行 Suspend Blocker(暂停阻滞)的更新，是因为在这里可能会释放 Suspend Blocker。

本阶段 Suspend Blocker 的更新很简单，仅需要判断 Device 是否需要持有 CPU 或者是否需要 CPU 继续运行，如果有没有释放的唤醒锁，或者还有用户活动的话，或者屏幕没有关闭的话等，则肯定是需要持有 CPU 的。所以这里就是根据需求去申请或者释放 SuspendBlocker。

到此为止，整个 PowerManagerService 的工作过程我们已经介绍完毕，已经基本上讲解了在 Framework 中的实现过程。

(4)　userActivity

如果要在屏幕中点击或者滑动一次就会调用一次 userActivity 函数，该函数会更新一些状态，其中比较重要的是 mUserState 值的状态，此值决定了系统对 LCD、按键以及键盘的亮和灭的处理，例如“mUserState = 0X7”表示 LCD 和按键背光亮。

函数 userActivity 的具体实现代码如下所示：

```
public void userActivity(long eventTime, int event, int flags) {
    final long now = SystemClock.uptimeMillis();
    if (mContext.checkCallingOrSelfPermission(
      android.Manifest.permission.DEVICE_POWER)
      != PackageManager.PERMISSION_GRANTED) {
        synchronized (mLock) {
            if (now >= mLastWarningAboutUserActivityPermission+(5*60*1000)) {
                mLastWarningAboutUserActivityPermission = now;
                Slog.w(TAG,
                  "Ignoring call to PowerManager.userActivity() because the "
                  + "caller does not have DEVICE_POWER permission.  "
                  + "Please fix your app!  "
                  + " pid=" + Binder.getCallingPid()
                  + " uid=" + Binder.getCallingUid());
            }
        }
        return;
    }
    if (eventTime > SystemClock.uptimeMillis()) {
        throw new IllegalArgumentException("event time must not be in the future");
    }
    final int uid = Binder.getCallingUid();
    final long ident = Binder.clearCallingIdentity();
    try {
        userActivityInternal(eventTime, event, flags, uid);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

(5) acquireWakeLock

函数 acquireWakeLock 的功能是获得唤醒锁，文件 PowerManagerService.java 的最基本功能是管理所有的应用程序申请的唤醒锁。函数 acquireWakeLock 的具体实现代码如下所示：

```
public void acquireWakeLock(
  IBinder lock, int flags, String tag, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    PowerManager.validateWakeLockParameters(flags, tag);

    mContext.enforceCallingOrSelfPermission(
      android.Manifest.permission.WAKE_ LOCK, null);
    if (ws!=null && ws.size()!=0) {
        mContext.enforceCallingOrSelfPermission(
          android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final int uid = Binder.getCallingUid();
    final int pid = Binder.getCallingPid();
    final long ident = Binder.clearCallingIdentity();
    try {
        acquireWakeLockInternal(lock, flags, tag, ws, uid, pid);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

在 Android 系统中，音频播放器、视频播放器、Camera 等申请的唤醒锁都是通过这个类来管理的。看下面的代码：

```
static final String PARTIAL_NAME = "PowerManagerService";
Power.acquireWakeLock(Power.PARTIAL_WAKE_LOCK, PARTIAL_NAME);
```

在上述代码中，调用了类 Power 中的函数 acquireWakeLock()，此时的 PARTIAL_NAME 作为参数传递到底层去。

(6) acquireWakeLockInternal

函数 acquireWakeLockInternal 的功能是获得唤醒的内部锁，具体实现代码如下所示：

```
private void acquireWakeLockInternal(IBinder lock, int flags, String tag,
  WorkSource ws, int uid, int pid) {
    synchronized (mLock) {
        if (DEBUG_SPEW) {
            Slog.d(TAG, "acquireWakeLockInternal: lock=" + Objects.hashCode(lock)
              + ", flags=0x" + Integer.toHexString(flags)
              + ", tag=\"" + tag + "\", ws=" + ws + ", uid=" + uid + ", pid=" + pid);
        }
```

```
        WakeLock wakeLock;
        int index = findWakeLockIndexLocked(lock);
        if (index >= 0) {
            wakeLock = mWakeLocks.get(index);
            if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {
                // Update existing wake lock. This shouldn't happen but is harmless.
                notifyWakeLockReleasedLocked(wakeLock);
                wakeLock.updateProperties(flags, tag, ws, uid, pid);
                notifyWakeLockAcquiredLocked(wakeLock);
            }
        } else {
            wakeLock = new WakeLock(lock, flags, tag, ws, uid, pid);
            try {
                lock.linkToDeath(wakeLock, 0);
            } catch (RemoteException ex) {
                throw new IllegalArgumentException("Wake lock is already dead.");
            }
            notifyWakeLockAcquiredLocked(wakeLock);
            mWakeLocks.add(wakeLock);
        }

        applyWakeLockFlagsOnAcquireLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;
        updatePowerStateLocked();
    }
}
```

(7) applyWakeLockFlagsOnAcquireLocked

函数 applyWakeLockFlagsOnAcquireLocked 的功能是在获取的锁中标志某个唤醒锁，具体实现代码如下所示：

```
private void applyWakeLockFlagsOnAcquireLocked(WakeLock wakeLock) {
    if ((wakeLock.mFlags&PowerManager.ACQUIRE_CAUSES_WAKEUP)!=0
      && isScreenLock(wakeLock)) {
        wakeUpNoUpdateLocked(SystemClock.uptimeMillis());
    }
}
```

(8) releaseWakeLock

函数 releaseWakeLock 的功能是释放唤醒锁，具体实现代码如下所示：

```
public void releaseWakeLock(IBinder lock, int flags) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    mContext.enforceCallingOrSelfPermission(
      android.Manifest.permission.WAKE_LOCK, null);
    final long ident = Binder.clearCallingIdentity();
    try {
```

```
        releaseWakeLockInternal(lock, flags);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

(9) releaseWakeLockInternal

函数 releaseWakeLockInternal 的功能是释放唤醒内部锁，具体实现代码如下所示：

```
private void releaseWakeLockInternal(IBinder lock, int flags) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "releaseWakeLockInternal: lock="
                  + Objects.hashCode(lock)
                  + " [not found], flags=0x" + Integer.toHexString(flags));
            }
            return;
        }

        WakeLock wakeLock = mWakeLocks.get(index);
        if (DEBUG_SPEW) {
            Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
              + " [" + wakeLock.mTag + "], flags=0x" + Integer.toHexString(flags));
        }

        mWakeLocks.remove(index);
        notifyWakeLockReleasedLocked(wakeLock);
        wakeLock.mLock.unlinkToDeath(wakeLock, 0);

        if ((flags&PowerManager.WAIT_FOR_PROXIMITY_NEGATIVE) != 0) {
            mRequestWaitForNegativeProximity = true;
        }

        applyWakeLockFlagsOnReleaseLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;
        updatePowerStateLocked();
    }
}
```

(10) updateWakeLockWorkSource 和 updateWakeLockWorkSourceInternal

函数 updateWakeLockWorkSource 和 updateWakeLockWorkSourceInternal 被绑定机制 Binder 所调用，功能是分别更新唤醒锁资源和内部唤醒锁的资源。

这两个函数的具体实现代码如下所示：

```
public void updateWakeLockWorkSource(IBinder lock, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
```

```
    }

    mContext.enforceCallingOrSelfPermission(
      android.Manifest.permission.WAKE_LOCK, null);
    if (ws!=null && ws.size()!=0) {
        mContext.enforceCallingOrSelfPermission(
          android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final long ident = Binder.clearCallingIdentity();
    try {
        updateWakeLockWorkSourceInternal(lock, ws);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}

private void updateWakeLockWorkSourceInternal(
  IBinder lock, WorkSource ws) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG,
                  "updateWakeLockWorkSourceInternal: lock="
                  + Objects.hashCode(lock)
                  + " [not found], ws=" + ws);
            }
            throw new IllegalArgumentException("Wake lock not active");
        }

        WakeLock wakeLock = mWakeLocks.get(index);
        if (DEBUG_SPEW) {
            Slog.d(TAG,
              "updateWakeLockWorkSourceInternal: lock="
              + Objects.hashCode(lock)
              + " [" + wakeLock.mTag + "], ws=" + ws);
        }

        if (!wakeLock.hasSameWorkSource(ws)) {
            notifyWakeLockReleasedLocked(wakeLock);
            wakeLock.updateWorkSource(ws);
            notifyWakeLockAcquiredLocked(wakeLock);
        }
    }
}
```

12.3 分析JNI层

在 Android 的电源管理系统中，在 Power 类中并没有实现 Native 申明的方法，而是在文件 frameworks/base/core/jni/android_os_Power.cpp 实现的。也就是说，文件 android_os_Power.cpp 就是 Power Management 系统的 JNI 层。在本节的内容中，将详细分析 Android 电源管理系统中的 JNI 层的基本知识。

12.3.1 文件android_os_Power.cpp

文件 android_os_Power.cpp 比较简单，定义了连接应用层和内核层之间的桥梁作用的接口函数，这些函数已经在本章前面的 Framework 层部分的内容中调用过。

文件 android_os_Power.cpp 的主要实现代码如下所示：

```
static void acquireWakeLock(JNIEnv *env, jobject clazz, jint lock, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
    acquire_wake_lock(lock, id);
    env->ReleaseStringUTFChars(idObj, id);
}
static void releaseWakeLock(JNIEnv *env, jobject clazz, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
    release_wake_lock(id);
    env->ReleaseStringUTFChars(idObj, id);
}
static int setLastUserActivityTimeout(JNIEnv *env, jobject clazz, jlong timeMS)
{
    return set_last_user_activity_timeout(timeMS/1000);
}
static int setScreenState(JNIEnv *env, jobject clazz, jboolean on)
{
    return set_screen_state(on);
}
static void android_os_Power_shutdown(JNIEnv *env, jobject clazz)
{
```

```
    android_reboot(ANDROID_RB_POWEROFF, 0, 0);
}
static void android_os_Power_reboot(JNIEnv *env, jobject clazz, jstring reason)
{
    if (reason == NULL) {
        android_reboot(ANDROID_RB_RESTART, 0, 0);
    } else {
        const char *chars = env->GetStringUTFChars(reason, NULL);
        android_reboot(ANDROID_RB_RESTART2, 0, (char*)chars);
        env->ReleaseStringUTFChars(reason, chars);  // In case it fails.
    }
    jniThrowIOException(env, errno);
}
static JNINativeMethod method_table[] = {
    { "acquireWakeLock", "(ILjava/lang/String;)V", (void*)acquireWakeLock },
    { "releaseWakeLock", "(Ljava/lang/String;)V", (void*)releaseWakeLock },
    { "setLastUserActivityTimeout", "(J)I", (void*)setLastUserActivityTimeout },
    { "setScreenState", "(Z)I", (void*)setScreenState },
    { "shutdown", "()V", (void*)android_os_Power_shutdown },
    { "rebootNative", "(Ljava/lang/String;)V", (void*)android_os_Power_reboot },
};
int register_android_os_Power(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(
        env, "android/os/Power",
        method_table, NELEM(method_table));
}
```

12.3.2　文件power.c

与 Linux Kernel 的交互工作是通过文件 hardware\libhardware\power\power.c 实现的，Andriod 跟 Kernel 的交互主要是通过 sys 文件的方式来实现的。

文件 power.c 的具体实现代码如下所示：

```
static void power_init(struct power_module *module)
{
}
static void power_set_interactive(struct power_module *module, int on)
{
}
static void power_hint(struct power_module *module, power_hint_t hint,
  void *data) {
    switch (hint) {
    default:
        break;
    }
}
```

```
static struct hw_module_methods_t power_module_methods = {
    .open = NULL,
};
struct power_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = POWER_MODULE_API_VERSION_0_2,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = POWER_HARDWARE_MODULE_ID,
        .name = "Default Power HAL",
        .author = "The Android Open Source Project",
        .methods = &power_module_methods,
    },
    .init = power_init,
    .setInteractive = power_set_interactive,
    .powerHint = power_hint,
};
```

12.4　分析Kernel(内核)层

在 Android 系统中，Power Management 系统内核层的实现文件如下：

- drivers/android/power.c
- drivers/base/main.c
- drivers/base/power/suspend.c
- drivers/base/power/resume.c
- kernel/power/main.c
- kernel/power/wakelock.c
- kernel/power/unwakelock.c
- kernel/power/earlysuspend.c

在本节的内容中，将对上述内核层文件的具体实现进行详细讲解。为了节省本书的篇幅，将只讲解主要的实现文件的核心代码。

12.4.1　文件power.c

在 Power Management 系统的内核层中，实现文件 drivers/android/power.c 对 Kernel 提供了如下所示的接口函数。

(1) EXPORT_SYMBOL(android_init_suspend_lock)

此接口函数的功能是初始化 Suspend lock，在使用 Power Management 内核之前，必须做初始化工作，函数 android_init_suspend_lock 的具体实现代码如下所示：

```
int android_init_suspend_lock(android_suspend_lock_t *lock) {
    return android_init_suspend_lock_internal(lock, 0);
}
```

(2) EXPORT_SYMBOL(android_uninit_suspend_lock)

此接口函数的功能是释放 suspend lock 相关的资源，函数 android_uninit_suspend_lock 的具体实现代码如下所示：

```
void android_uninit_suspend_lock(android_suspend_lock_t *lock)
{
    unsigned long irqflags;
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_uninit_suspend_lock name=%s\n",
            lock->name);
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(lock->stat.count) {
        if(g_deleted_wake_locks.stat.count == 0) {
            g_deleted_wake_locks.name = "deleted_wake_locks";
            android_init_suspend_lock_internal(
                &g_deleted_wake_locks, 1);
        }
        g_deleted_wake_locks.stat.count += lock->stat.count;
        g_deleted_wake_locks.stat.expire_count += lock->stat.expire_count;
        g_deleted_wake_locks.stat.total_time =
          ktime_add(
           g_deleted_wake_locks.stat.total_time, lock->stat.total_time);
        g_deleted_wake_locks.stat.max_time =
          ktime_add(g_deleted_wake_locks.stat.max_time, lock->stat.max_time);
    }
#endif
    list_del(&lock->link);
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

(3) EXPORT_SYMBOL(android_lock_suspend)

此接口函数的功能是申请 lock，并调用相应的 unlock 来释放 lock。函数 android_lock_suspend 的具体实现代码如下所示：

```
void android_lock_suspend(android_suspend_lock_t *lock)
{
    unsigned long irqflags;
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
        lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
        lock->stat.last_time = ktime_get();
    }
#endif
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_power: acquire wake lock: %s\n",
            lock->name);
    lock->expires = INT_MAX;
```

```
    lock->flags &= ~ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
    list_del(&lock->link);
    list_add(&lock->link, &g_active_partial_wake_locks);
    g_current_event_num++;
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

(4) EXPORT_SYMBOL(android_lock_suspend_auto_expire)

此接口函数的功能是申请 Partial Wakelock，当定时时间到后，会自动释放。

函数 android_lock_suspend_auto_expire 的具体实现代码如下所示：

```
void android_lock_suspend_auto_expire(android_suspend_lock_t *lock, int timeout)
{
    unsigned long irqflags;
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
        lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
        lock->stat.last_time = ktime_get();
    }
#endif
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_power: acquire wake lock: %s, "
            "timeout %d.%03lu\n", lock->name, timeout / HZ,
            (timeout % HZ) * MSEC_PER_SEC / HZ);
    lock->expires = jiffies + timeout;
    lock->flags |= ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
    list_del(&lock->link);
    list_add(&lock->link, &g_active_partial_wake_locks);
    g_current_event_num++;
    wake_up(&g_wait_queue);
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

(5) EXPORT_SYMBOL(android_unlock_suspend)

此接口函数的功能是释放 lock，函数 android_unlock_suspend 的具体实现代码如下所示：

```
static void android_unlock_suspend_stat_locked(android_suspend_lock_t *lock)
{
    if(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE) {
        ktime_t duration;
        lock->flags &= ~ANDROID_SUSPEND_LOCK_ACTIVE;
        lock->stat.count++;
        duration = ktime_sub(ktime_get(), lock->stat.last_time);
        lock->stat.total_time = ktime_add(lock->stat.total_time, duration);
        if(ktime_to_ns(duration) > ktime_to_ns(lock->stat.max_time))
            lock->stat.max_time = duration;
        lock->stat.last_time = ktime_get();
    }
}
```

(6)　EXPORT_SYMBOL(android_register_early_suspend)

此接口函数的功能是注册 Early Suspend 的驱动，函数 android_register_early_suspend 的具体实现代码如下所示：

```
void android_register_early_suspend(android_early_suspend_t *handler)
{
    struct list_head *pos;

    mutex_lock(&g_early_suspend_lock);
    list_for_each(pos, &g_early_suspend_handlers) {
        android_early_suspend_t *e=list_entry(pos,android_early_suspend_t,link);
        if(e->level > handler->level)
            break;
    }
    list_add_tail(&handler->link, pos);
    mutex_unlock(&g_early_suspend_lock);
}
```

(7)　EXPORT_SYMBOL(android_unregister_early_suspend)

此接口函数的功能是取消已经注册的 Early Suspend 的驱动，函数 android_unregister_early_suspend 的具体实现代码如下所示：

```
void android_unregister_early_suspend(android_early_suspend_t *handler)
{
    mutex_lock(&g_early_suspend_lock);
    list_del(&handler->link);
    mutex_unlock(&g_early_suspend_lock);
}
```

12.4.2　文件earlysuspend.c

在 Power Management 系统的内核层中，实现文件 kernel/power/earlysuspend.c 对 Kernel 提供了如下所示的接口函数。

(1)　EXPORT_SYMBOL(register_early_suspend)

此接口函数的功能是注册 Early Suspend 的驱动，函数 register_early_suspend 的具体实现代码如下所示：

```
void register_early_suspend(struct early_suspend *handler)
{
    struct list head *pos;

    mutex_lock(&early_suspend_lock);
    list_for_each(pos, &early_suspend_handlers) {
        struct early_suspend *e;
        e = list_entry(pos, struct early_suspend, link);
        if (e->level > handler->level)
            break;
```

```
    }
    list add tail(&handler->link, pos);
    if ((state&SUSPENDED) && handler->suspend)
        handler->suspend(handler);
    mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(register_early_suspend);
```

(2) EXPORT_SYMBOL(unregister_early_suspend)

此接口函数的功能是取消已经注册的 Early Suspend 的驱动，函数 unregister_early_suspend 的具体实现代码如下所示：

```
void unregister_early_suspend(struct early_suspend *handler)
{
    mutex_lock(&early_suspend_lock);
    list_del(&handler->link);
    mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(unregister_early_suspend);
```

12.4.3 文件wakelock.c

在 Power Management 系统的内核层中，实现文件 kernel/power/wakelock.c 对 Kernel 提供了如下所示的接口函数。

(1) EXPORT_SYMBOL(wake_unlock)

此接口函数的功能是释放 lock，函数 wake_unlock 的具体实现代码如下所示：

```
void wake_unlock(struct wake_lock *lock)
{
    int type;
    unsigned long irqflags;
    spin_lock_irqsave(&list_lock, irqflags);
    type = lock->flags & WAKE_LOCK_TYPE_MASK;
#ifdef CONFIG_WAKELOCK_STAT
    wake_unlock_stat_locked(lock, 0);
#endif
    if (debug_mask & DEBUG_WAKE_LOCK)
        pr info("wake unlock: %s\n", lock->name);
    lock->flags &= ~(WAKE_LOCK_ACTIVE | WAKE_LOCK_AUTO_EXPIRE);
    list_del(&lock->link);
    list add(&lock->link, &inactive locks);
    if (type == WAKE_LOCK_SUSPEND) {
        long has_lock = has_wake_lock_locked(type);
        if (has_lock > 0) {
            if (debug_mask & DEBUG_EXPIRE)
                pr info("wake unlock: %s, start expire timer, ",
                "%ld\n", lock->name, has_lock);
            mod_timer(&expire_timer, jiffies + has_lock);
```

```
        } else {
            if (del timer(&expire timer))
                if (debug_mask & DEBUG_EXPIRE)
                    pr_info("wake_unlock: %s, stop expire ",
                    "timer\n", lock->name);
            if (has_lock == 0)
                queue_work(suspend_work_queue, &suspend_work);
        }
        if (lock == &main_wake_lock) {
            if (debug_mask & DEBUG_SUSPEND)
                print_active_locks(WAKE_LOCK_SUSPEND);
#ifdef CONFIG_WAKELOCK_STAT
            update_sleep_wait_stats_locked(0);
#endif
        }
    }
    spin_unlock_irqrestore(&list_lock, irqflags);
}
EXPORT_SYMBOL(wake_unlock);
```

(2)　EXPORT_SYMBOL(wake_lock)

此接口函数的功能是申请 lock，必须调用相应的 unlock 来释放 lock。函数 wake_lock 的具体实现代码如下所示：

```
void wake lock(struct wake lock *lock)
{
    wake_lock_internal(lock, 0, 0);
}
EXPORT_SYMBOL(wake_lock);
```

(3)　static DEFINE_TIMER(expire_timer, expire_wake_locks, 0, 0)

此接口函数的功能是，如果定时时间到，则加入到 suspend 队列中。函数 expire_wake_locks 的具体实现代码如下所示：

```
static void expire_wake_locks(unsigned long data)
{
    long has_lock;
    unsigned long irqflags;
    if (debug_mask & DEBUG_EXPIRE)
        pr_info("expire_wake_locks: start\n");
    spin_lock_irqsave(&list_lock, irqflags);
    if (debug_mask & DEBUG_SUSPEND)
        print_active_locks(WAKE_LOCK_SUSPEND);
    has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
    if (debug_mask & DEBUG_EXPIRE)
        pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
    if (has_lock == 0)
        queue_work(suspend_work_queue, &suspend_work);
    spin_unlock_irqrestore(&list_lock, irqflags);
```

```
}
```

12.4.4 文件resume.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/resume.c 对 Kernel 提供了如下所示的接口函数。

(1) EXPORT_SYMBOL_GPL(device_power_up)

此接口函数的功能是打开特殊的设备，函数 device_power_up 的具体实现代码如下所示：

```
void device_power_up(void)
{
    sysdev resume();
    dpm_power_up();
}
EXPORT_SYMBOL_GPL(device_power_up);
```

(2) EXPORT_SYMBOL_GPL(device_resume)

此接口函数的功能是重新存储设备的状态，函数 device_resume 的具体实现代码如下所示：

```
void device_resume(void)
{
    down(&dpm sem);
    dpm_resume();
    up(&dpm_sem);
}
EXPORT_SYMBOL_GPL(device_resume);
```

12.4.5 文件suspend.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/suspend.c 对 Kernel 提供了如下所示的接口函数。

(1) EXPORT_SYMBOL_GPL(device_suspend)

此接口函数的功能是保存系统状态，并结束系统中的设备的运行。函数 device_suspend 的具体实现代码如下所示：

```
int device_suspend(pm_message_t state)
{
    int error = 0;
    down(&dpm_sem);
    down(&dpm_list_sem);
    while (!list_empty(&dpm_active) && error==0) {
        struct list_head *entry = dpm_active.prev;
        struct device *dev = to_device(entry);
        get_device(dev);
        up(&dpm_list_sem);
        error = suspend_device(dev, state);
        down(&dpm_list_sem);
```

```
        /* Check if the device got removed */
        if (!list empty(&dev->power.entry)) {
            /* Move it to the dpm_off or dpm_off_irq list */
            if (!error) {
                list_del(&dev->power.entry);
                list_add(&dev->power.entry, &dpm_off);
            } else if (error == -EAGAIN) {
                list_del(&dev->power.entry);
                list_add(&dev->power.entry, &dpm_off_irq);
                error = 0;
            }
        }
        if (error)
            printk(KERN_ERR "Could not suspend device %s: "
                    "error %d\n", kobject_name(&dev->kobj), error);
        put_device(dev);
    }
    up(&dpm_list_sem);
    if (error)
        dpm_resume();
    up(&dpm sem);
    return error;
}
EXPORT_SYMBOL_GPL(device_suspend);
```

(2)　EXPORT_SYMBOL_GPL(device_power_down)

此接口函数的功能是关闭特殊设备，函数 device_power_down 的具体实现代码如下所示：

```
int device_power_down(pm_message_t state)
{
    int error = 0;
    struct device *dev;
    list_for_each_entry_reverse(dev, &dpm_off_irq, power.entry) {
        if ((error = suspend_device(dev, state)))
            break;
    }
    if (error)
        goto Error;
    if ((error = sysdev_suspend(state)))
        goto Error;

 Done:
    return error;

 Error:
    dpm_power_up();
    goto Done;
}
EXPORT_SYMBOL_GPL(device_power_down);
```

12.4.6　文件main.c

在 Power Management 系统的内核层中，内核文件 kernel/power/main.c 的主要实现代码如下所示：

```
static int __init pm_init(void)
{
    int error = pm_start_workqueue();
    if (error)
        return error;
    hibernate_image_size_init();
    hibernate_reserved_size_init();
    power_kobj = kobject_create_and_add("power", NULL);
    if (!power_kobj)
        return -ENOMEM;
    return sysfs_create_group(power_kobj, &attr_group);
}
core_initcall(pm_init);
```

在上述代码中，函数 pm_init(void)的返回值为 sysfs_create_group(power_kobj, &attr_group)，表示当我们对 sysfs/目录下相对的节点进行操作时会调用 attr_group 中的相关函数。

12.4.7　proc文件

(1) 在 Power Management 系统的内核层中，给 Framework 层提供了如下所示的 proc 文件。

- /sys/android_power/acquire_partial_wake_lock：申请 partial wake lock。
- /sys/android_power/acquire_full_wake_lock：申请 full wakelock。
- /sys/android_power/release_wake_lock：释放相应的 wake lock。
- /sys/android_power/request_state：请求改变系统状态，进 standby 和回到 wakeup 两种状态。
- /sys/android_power/state：指示当前系统的状态。

(2) 在 Android 电源管理系统中的主要功能是通过唤醒锁来实现的，在最底层，主要是通过如下三个队列来实现其管理功能：

- static LIST_HEAD(g_inactive_locks)。
- static LIST_HEAD(g_active_partial_wake_locks)。
- static LIST_HEAD(g_active_full_wake_locks)。

(3) 在处理过程中实现如下所示的插入和移动操作：

- 所有被初始化的 lock 都会被插入到队列 g_inactive_locks 中。
- 当前活动的 Partial Wake Lock 被插入到 g_active_partial_wake_locks 队列中。
- 活动的 Full Wake Lock 被插入到 g_active_full_wake_locks 队列中。
- 对于所有的 Partial Wake Lock 和 Full Wake Lock，在过期后或 unlock 后，都会被移到 inactive 队列中，以等待下次被调用。

12.5　wakelock和early_suspend

在 Android 系统中，wakelock 和 early_suspend 是一种特殊机制，能够实现系统的“唤醒”和“休眠”功能，获取系统资源的信息，例如电源信息和 CPU 信息等。在本节的内容中，将详细讲解 wakelock 和 early_suspend 机制的基本知识。

12.5.1　wakelock的原理

wakelock 在 Android 的电源管理系统中扮演一个核心的角色。wakelock 是一种“锁”机制，只要有人拿着这个锁，系统就无法进入休眠状态。这个锁可以是有超时的或者是没有超时的，超时的锁会在时间过去以后自动解锁。如果没有锁了或者超时了，内核就会启动休眠的那套机制来进入休眠。

当系统启动完毕后，会自己去加一把名为 main 的锁，而当系统有意愿去睡眠时，则会先去释放这把 main 锁。在 Android 中，在 early_suspend 的最后一步会去释放 main 锁(wake_unlock: main)。释放完后，则会去检查是否还有其他存在的锁，如果没有，则直接进入睡眠过程。

它的缺点是，如果有某一应用获锁而不释放，或者因一直在执行某种操作而没时间来释放的话，则会导致系统一直进入不了睡眠状态，功耗过大。

在 wakelock 中有 3 种类型，最常用的是 WAKE_LOCK_SUSPEND，作用是防止系统进入睡眠。wakelock 的接口定义在文件 wakelock.c 中，定义代码如下所示：

```
enum {
    WAKE_LOCK_SUSPEND, /* Prevent suspend */
    WAKE_LOCK_IDLE,    /* Prevent low power idle */
    WAKE_LOCK_TYPE_COUNT
};
```

在 wakelock 中，有如下两个地方可以让系统从 early_suspend 进入 suspend 状态：

- 在 wake_unlock 中，当解锁时，如果没有其他的 wakelock，则进入 suspend。
- 当超时锁的定时器超时后，定时器的回调函数会判断有没有其他的 wakelock，若没有，则进入 suspend。

在 Android 系统中，在 Kernel 层使用 wakelock 的基本步骤如下所示。

(1) 调用函数 android_init_suspend_lock 初始化一个 wakelock。

(2) 调用相关申请 lock 的函数 android_lock_suspend 或 android_lock_suspend_auto_expire 请求 lock，此处只能申请 Partial Wake Lock，如果要申请 Full Wake Lock，则需要调用函数 android_lock_partial_suspend_auto_expire(该函数没有 EXPORT 出来)。

(3) 如果是自动超时的 wakelock，则可忽略，否则必须及时把相关的 wakelock 释放掉，否则会造成系统长期运行在高功耗的状态。

(4) 在驱动卸载或不再使用 wakelock 时，需要及时地调用 android_uninit_suspend_lock 以释放资源。

由此可以总结出，Kernel 的 wakelock 唤醒操作的基本顺序依次如下。

① 调用框架层函数 acquireWakeLock()，此函数的具体实现代码如下所示：

```
int acquire_wake_lock(int lock, const char *id)
{
    initialize_fds();
    // LOGI("acquire_wake_lock lock=%d id='%s'\n", lock, id);
    if (g_error) return g_error;
    int fd;
    if (lock == PARTIAL_WAKE_LOCK) {
        fd = g_fds[ACQUIRE_PARTIAL_WAKE_LOCK];
    }
    else {
        return EINVAL;
    }
    return write(fd, id, strlen(id));
}
```

② 调用 android_os_Power.cpp 的 acquireWakeLock()。

③ 调用 power.c 的 acquire_wake_lock()。

12.5.2 early_suspend的原理

early_suspend 在 Linux 内核的睡眠过程前被调用。因为背光需要的能耗过大，所以常采用此类方法在手机系统的设计中操作背光。如在一些在内核中的预先进行处理的事件可以先注册上 early_suspend 函数，这样，当系统要进入睡眠之前，会首先调用这些注册的函数。

与 Android 休眠唤醒相关的实现文件如下所示：

```
linux_source/kernel/power/main.c
linux source/kernel/power/earlysuspend.c
linux_source/kernel/power/wakelock.c
linux_source/kernel/power/process.c
linux_source/driver/base/power/main.c
linux_source/arch/xxx/mach-xxx/pm.c 或 linux_source/arch/xxx/plat-xxx/pm.c
```

12.5.3 Android休眠

当用户读写/sys/power/state 时，文件 linux_source/kernel/power/main.c 中的 state_store()函数会被调用。其中，Android 的 early_suspend 会执行：

```
request_suspend_state(state);
```

标准的 Linux 休眠会执行：

```
error = enter_state(state);
```

函数 state_store()的原型如下所示：

```
static ssize_t state_store(struct kobject *kobj, struct kobj_attribute *attr,
                           const char *buf, size_t n)
```

在函数 request_suspend_state()中，会调用 early_suspend_work 的工作队列，以进入 early_suspend()函数中。函数 request_suspend_state()的原型如下所示：

```
void request_suspend_state(suspend_state_t new_state)
```

在函数 early_suspend()中，首先要判断当前请求的状态是否还是 suspend，如果不是，则直接退出；如果是，则函数会调用已经注册的 early_suspend 的函数。然后同步文件系统，最后释放 main_wake_lock。函数 early_suspend()的原型如下所示：

```
static void early_suspend(struct work_struct *work)
```

在函数 wake_unlock()中删除链表中的 wake_lock 节点，以判断当前是否存在 wake_lock。如果 wake_lock 的数目为 0，则调用工作队列 suspend_work，然后进入 suspend 状态。

函数 wake_unlock()的原型如下所示：

```
void wake_unlock(struct wake_lock *lock)
```

在函数 suspend()中，首先判断当前是否有 wake_lock，如果有则退出；然后同步文件系统，最后调用 pm_suspend()函数。函数 suspend()的原型如下所示：

```
static void suspend(struct work_struct *work)
```

在函数 pm_suspend()中调用 enter_state()函数，这样就进入了标准 Linux 的休眠过程。函数 pm_suspend()的原型如下所示：

```
int pm_suspend(suspend_state_t state)
```

在函数 enter_state()中，首先检查一些状态参数，再同步文件系统，然后调用 suspend_prepare()来冻结进程，最后调用 suspend_devices_and_enter()让外设进入休眠。函数 enter_state()的原型如下所示：

```
static int enter_state(suspend_state_t state)
```

在函数 suspend_prepare()中，先通过“pm_prepare_console();”给 suspend 分配一个虚拟终端来输出信息，再广播一个系统进入 suspend 的通报，关闭用户态的 helper 进程，然后调用函数 suspend_freeze_processes()来冻结进程，最后会尝试释放一些内存。

函数 suspend_prepare()的原型如下所示：

```
static int suspend_prepare(void)
```

在函数 suspend_freeze_processes()中调用 freeze_processes()函数，而在 freeze_processes()函数中又调用了 try_to_freeze_tasks()函数来完成冻结任务。在冻结过程中，会判断当前进程是否有 wake_lock，如果有，则冻结失败，函数会放弃冻结。

函数 freeze_processes()的原型如下所示：

```
static int try_to_freeze_tasks(bool sig_only)
```

到此为止，所有的进程都已经停止了，内核态进程有可能在停止的时候握有一些信号量，如果这时在外设里面去解锁这个信号量，有可能会发生死锁，所以建议不要在外设的 suspend()里面等待锁。而且在 suspend 的过程中，很多 log 是无法输出的，所以一旦出现问题，就非常

难以调试。

接下来回到 enter_state()函数中，当冻结进程完成后，调用 suspend_devices_and_enter()函数，目的是让外设进入休眠。在该函数中，首先休眠串口，然后通过 device_suspend()函数调用各驱动的 suspend 函数。

当外设进入休眠后，调用 suspend_ops->prepare()，suspend_ops 是板级的 PM 操作，假如是 s3c6410，则被注册在文件 linux_source/arch/arm/plat-s3c64xx/pm.c 中，在里面只定义了 suspend_ops->enter()函数：

```
static struct platform suspend ops s3c6410 pm ops = {
    .enter       = s3c6410_pm_enter,
    .valid       = suspend_valid_only_mem,
};
```

然后在多 CPU 中关闭非启动 CPU，具体代码如下所示：

```
int suspend_devices_and_enter(suspend_state_t state)
{
    int error;
    if (!suspend ops)
        return -ENOSYS;
    if (suspend_ops->begin) {
        error = suspend_ops->begin(state);
        if (error)
            goto Close;
    }
    suspend_console();
    suspend test start();
    error = device_suspend(PMSG_SUSPEND);
    if (error) {
        printk(KERN_ERR "PM: Some devices failed to suspend\n");
        goto Recover_platform;
    }
    suspend_test_finish("suspend devices");
    if (suspend_test(TEST_DEVICES))
        goto Recover_platform;
    if (suspend_ops->prepare) {
        error = suspend_ops->prepare();
        if (error)
            goto Resume_devices;
    }
    if (suspend_test(TEST_PLATFORM))
        goto Finish;
    error = disable_nonboot_cpus();
    if (!error && !suspend_test(TEST_CPUS))
        suspend_enter(state);
    enable nonboot cpus();
 Finish:
    if (suspend_ops->finish)
```

```
        suspend_ops->finish();
Resume devices:
    suspend_test_start();
    device_resume(PMSG_RESUME);
    suspend_test_finish("resume devices");
    resume_console();
Close:
    if (suspend_ops->end)
        suspend_ops->end();
    return error;
Recover_platform:
    if (suspend_ops->recover)
        suspend_ops->recover();
    goto Resume_devices;
}
```

接下来调用函数 suspend_enter()，该函数将首先关闭 IRQ，然后调用 device_power_down()，此函数会调用 suspend_late()函数。这个函数是系统真正进入休眠最后调用的函数，通常会在这个函数中做最后的检查，接下来休眠所有的系统设备和总线。最后调用 suspend_pos->enter()使 CPU 进入省电状态。此时整个休眠过程完成了。函数 suspend_enter()的原型如下所示：

```
static int suspend_enter(suspend_state_t state)
```

12.5.4　Android唤醒

如果在休眠中系统被中断或者有其他事件唤醒，接下来的代码就从 suspend 完成的地方开始执行，以 s3c6410 为例，即为文件 pm.c 中的 s3c6410_pm_enter()中的 cpu_init()，然后执行 suspend_enter()的 sysdev_resume()函数，唤醒系统设备和总线，启用系统中断。然后回到 suspend_devices_and_enter()函数中，启用休眠时停止掉的非启动 CPU，并且继续唤醒每个设备。当函数 suspend_devices_and_enter()被执行完成后，系统外设已经唤醒，但进程依然处于冻结的状态，返回到 enter_state 函数中，调用 suspend_finish()函数。在函数 suspend_finish()中解冻进程和任务，广播一个系统从 suspend 状态退出的 notify。

当所有的唤醒已经结束后，用户进程都已经开始运行了，但没点亮屏幕，唤醒通常会是以下的几种原因。

(1) 如果是来电，那么 Modem 会发送命令给 rild，这样可以让 rild 通知 WindowManager 有来电响应，这样就会远程调用 PowerManagerService 来写 on 到/sys/power/state 以调用 late resume()，执行点亮屏幕等操作。

(2) 用户按键事件会送到 WindowManager 中，WindowManager 会处理这些按键事件，按键分为几种情况，如果按键不是唤醒键，那么 WindowManager 会主动放弃 wakeLock 来使系统再次进入休眠；如果按键是唤醒键，那么 WindowManger 就会调用 PowerManagerService 中的接口来执行 late Resume。

(3) 当 on 被写入到/sys/power/state 后，与 early_suspend 过程一样，request_suspend_state()会被调用，只是执行的工作队列变为 late_resume_work。在 late_resume()函数中，唤醒调用了 early_suspend 的设备。

第 13 章 输入系统驱动应用

Android 输入系统的结构比较简单，实现输入功能的硬件设备包括键盘、触摸屏和轨迹球等。在 Android 的上层结构中，可以获得这些设备产生的事件，并对设备的事件做出响应。在 Java 框架中，通常使用运动事件来获得触摸屏和轨迹球设备的信息，使用按键事件获得各种键盘的信息。

本章将详细讲解 Android 4.3 输入系统的基本知识，介绍输入系统的源码和实现原理，为读者步入本书后面知识的学习打下基础。

13.1　输入系统介绍

Android 输入系统的基本框架结构如图 13-1 所示。

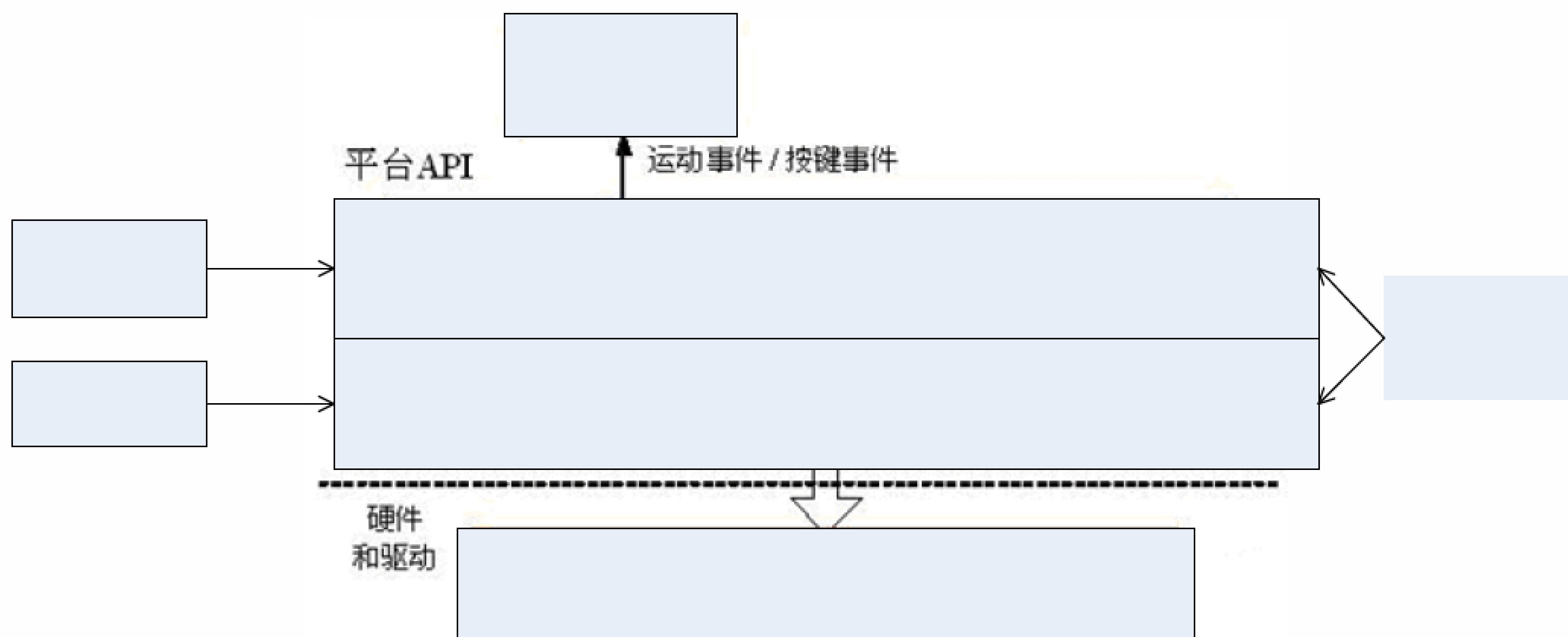


图 13-1　Android输入系统的框架结构

Android 输入系统的结构比较简单，自下而上包含了驱动程序、本地库处理部分、Java 类对输入事件的处理和 Java 程序的接口等，如图 13-2 所示。

图 13-2 中，从顶层到底层各个结构元素的具体说明如下所示。

(1)　Android 应用程序层

在 Android 系统的应用程序层中，通过重新实现 onTouchEvent 和 onTrackballEvent 等函数来接收运动事件(MotionEvent)，通过重新实现 onKeyDown 和 onKeyUp 等函数来接收按键事件(KeyEvent)。这些类包含在 android.view 包中。

(2)　Java 框架层的处理

在 Android 系统的 Java 框架层中，通过 KeyInputDevice 等类处理由 EventHub 传送上来的信息，这些信息通常由数据结构 RawInputEvent 和 KeyEvent 来表示。在一般情况下，对于按键事件，直接使用 KeyEvent 来传送给应用程序层。

对于触摸屏和轨迹球等事件，则由 RawInputEvent 经过转换后形成 MotionEvent 事件传送给应用程序层。

(3)　EventHub

在 Android 系统中，本地框架层的 EventHub 是 libui 中的一部分，它实现了对驱动程序的控制，并从中获得信息。定义按键布局和按键字符映射需要运行时配置文件的支持，它们的后缀名分别为“kl”和“kcm”。

(4)　驱动程序

在 Android 系统中，输入系统的驱动程序保存在/dev/input 目录中，通常是 Event 类型的驱动程序。

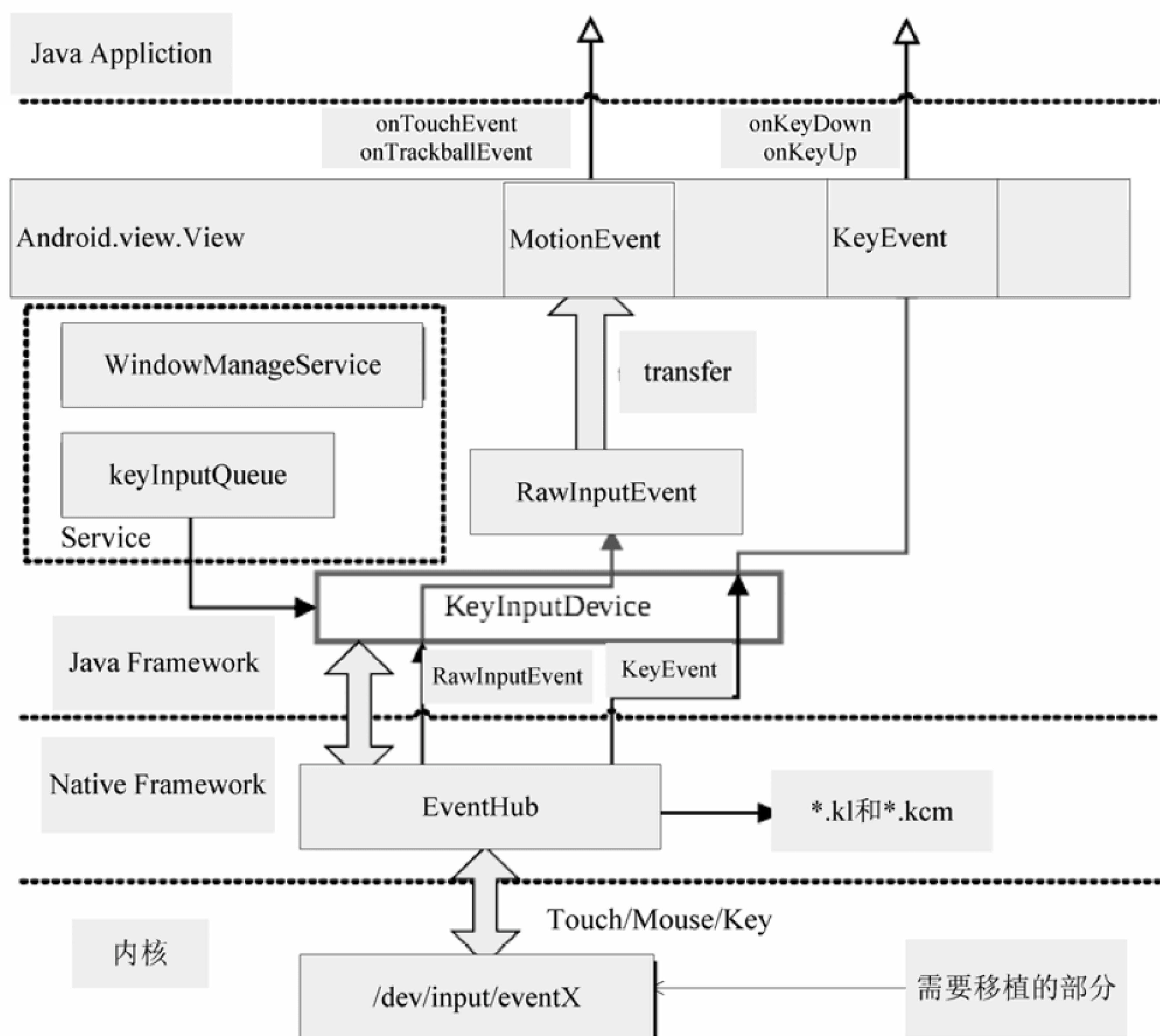


图 13-2　用户输入系统的结构

在 Android 系统中，Input 输入子系统的架构如图 13-3 所示。

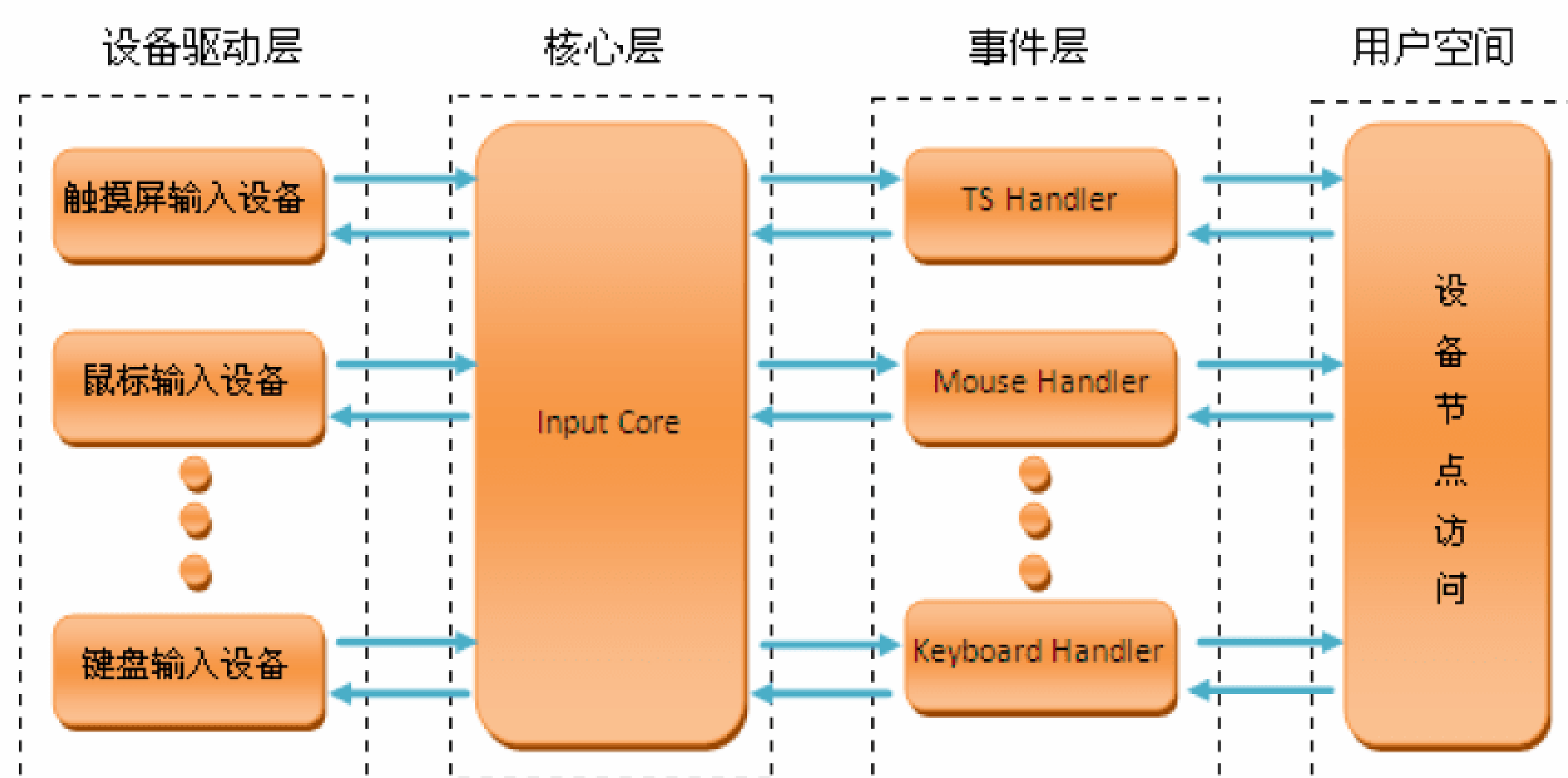


图 13-3　Input输入子系统的架构

13.2 分析Input(输入)系统驱动

在 Android 系统中，Input(输入)驱动程序是 Linux 输入设备的驱动程序，可以进一步分成游戏杆(joystick)、鼠标(mouse 和 mice)和事件队列(Event queue)三种驱动程序。事件驱动程序是目前通用的驱动程序，可以支持键盘、鼠标、触摸屏等多种输入设备。

Input 驱动程序的主设备号是 13，每一种 Input 设备占用 5 位，因此每种设备包含的个数是 32 个。Event 设备在用户空间大多使用如下三种文件系统来操作接口。

- Read：用于读取输入信息。
- Ioctl：用于获得和设置信息。
- Poll：调用可以进行用户空间的阻塞，当内核有按键等中断时，通过在中断中唤醒 poll 的内核实现，这样在用户空间的 poll 调用也可以返回。

Event 设备在文件系统中的设备节点为/dev/input/eventX 目录。主设备号为 13，次设备号按照递增顺序生成，为 64~95，各个具体的设备保存在 misc、touchscreen 和 keyboard 等目录中。Android 输入设备驱动程序的头文件是 include/linux/input.h，核心文件是 drivers/input/input.c，Event 部分的代码文件是 drivers/input/evdev.c。

13.2.1 分析头文件

(1) 首先看按键数值的定义，因为在 Android 手机系统中使用的键盘(keyboard)和小键盘(kaypad)属于按键设备 EV_KEY，轨迹球属于相对设备 EV_REL，触摸屏属于绝对设备 EV_ABS。在文件 input.h 中定义按键数值的代码如下所示：

```
#define KEY_RESERVED  0
#define KEY_ESC        1
#define KEY_1          2
#define KEY_2          3
#define KEY_3          4
#define KEY_4          5
#define KEY_5          6
#define KEY_6          7
#define KEY_7          8
#define KEY_8          9
#define KEY_9          10
#define KEY_0          11
#define KEY_MINUS       12
#define KEY_EQUAL       13
#define KEY_BACKSPACE  14
#define KEY_TAB         15
#define KEY_Q          16
#define KEY_W          17
#define KEY_E          18
#define KEY_R          19
```

```
#define KEY_T          20
```

(2) 然后定义结构体 input_dev，功能是表示 Input 驱动程序的各种信息，在里面定义并归纳了各种设备的信息，例如按键、相对设备、绝对设备、杂项设备、LED、声音设备，强制反馈设备、开关设备等。结构 struct input_dev 的定义代码如下所示：

```
struct input_dev {
    const char *name;              // 设备名称
    const char *phys;              // 设备在系统中的物理路径
    const char *uniq;              // 统一的 ID
    struct input_id id;            // 设备 ID
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];  // 事件
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; // 按键
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; // 相对设备
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; // 绝对设备
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; // 杂项设备
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; // LED
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; // 声音设备
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)]; // 强制反馈设备
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; // 开关设备
    unsigned int keycodemax;       // 按键码的最大值
    unsigned int keycodesize;      // 按键码的大小
    void *keycode;                 // 按键码
    int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
    int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
    struct ff_device *ff;
    unsigned int repeat_key;
    struct timer_list timer;
    int sync;
    int abs[ABS_MAX + 1];
    int rep[REP_MAX + 1];
    unsigned long key[BITS_TO_LONGS(KEY_CNT)];
    unsigned long led[BITS_TO_LONGS(LED_CNT)];
    unsigned long snd[BITS_TO_LONGS(SND_CNT)];
    unsigned long sw[BITS_TO_LONGS(SW_CNT)];
    int absmax[ABS_MAX + 1];       // 绝对设备相关内容
    int absmin[ABS_MAX + 1];
    int absfuzz[ABS_MAX + 1];
    int absflat[ABS_MAX + 1];      // 设备相关的操作
    int (*open)(struct input_dev *dev);
    void (*close)(struct input_dev *dev);
    int (*flush)(struct input_dev *dev, struct file *file);
    int (*event)(struct input_dev *dev, unsigned int type,
             unsigned int code, int value);
    struct input_handle *grab;
    spinlock_t event_lock;
    struct mutex mutex;
    unsigned int users;
    int going_away;
```

```
    struct device dev;
    struct list head  h list;
    struct list_head  node;
    unsigned int num_vals;
    unsigned int max_vals;
    struct input_value *vals;
    bool devres_managed;
};
```

(3) 在具体实现 Event 驱动程序时，可以使用接口通过向上通知的方式得到按键的事件。在文件 input.h 中定义实现上述接口的代码，如下所示：

```
384 void input_event(struct input_dev *dev, unsigned int type, unsigned int code,
int value);
385 void input_inject_event(struct input_handle *handle, unsigned int type,
unsigned int code, int value);
386
387 static inline void input_report_key(struct input_dev *dev, unsigned int code,
int value)
388 {
389     input_event(dev, EV_KEY, code, !!value);
390 }
391
392 static inline void input_report_rel(struct input_dev *dev, unsigned int code,
int value)
393 {
394     input_event(dev, EV_REL, code, value);
395 }
396
397 static inline void input_report_abs(struct input_dev *dev, unsigned int code,
int value)
398 {
399     input_event(dev, EV_ABS, code, value);
400 }
401
402 static inline void input report ff status(struct input dev *dev,
unsigned int code, int value)
403 {
404     input_event(dev, EV_FF_STATUS, code, value);
405 }
406
407 static inline void input_report_switch(struct input_dev *dev,
unsigned int code, int value)
408 {
409     input_event(dev, EV_SW, code, !!value);
410 }
411
412 static inline void input_sync(struct input_dev *dev)
413 {
```

```
414     input_event(dev, EV_SYN, SYN_REPORT, 0);
415 }
416
417 static inline void input_mt_sync(struct input_dev *dev)
418 {
419     input_event(dev, EV_SYN, SYN_MT_REPORT, 0);
420 }
```

(4)　基于文件 input.c 的原理，下面是作者编写的 USB 输入驱动程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/poll.h>
#include <linux/input.h>

#define USB_MOUSE   ("/dev/input/mouse0")

struct pollfd  mypoll;
int main(int argc,  char *argv[])
{
    int mouseFd;
    struct input_event buff;

    if ((mouseFd=open(USB_MOUSE, O_RDONLY)) == -1) {
        printf("Failed to open /dev/input/mouse0\n");
        return -1;
    }
    mypoll.fd =  mouseFd;
    mypoll.events = POLLIN;
    while(1)
    {
        if(poll(&mypoll, 1, 10) > 0)
        {
            unsigned char data[4] = {0};
            /*
            data 的数据格式：
            data0:00xx 1xxx  ----低三位是按键值---左中右分别为 01 02 04，
            第 4/5 位分别代表 x、y 移动方向，右上方 x/y>0，左下方 xy<0
            data1:取值范围-127~127，代表 x 轴移动偏移量
            data2:取值范围-127~127，代表 y 轴移动偏移量
            */
            usleep(50000);
            //MOUSEDEV_EMUL_PS2 方式每次采样数据为 3 个字节，多读不会出错，
            //只返回成功读取的数据数
            read(mouseFd, data, 4);
            printf("mouse data=%02x%02x%02x%02x\n",
              data[0],data[1], data[2], data[3]);
        }
```

```
    }
    close(mouseFd);
    return 0;
}
```

(5) 再看结构体 ff_device，表示强制反馈设备的数据，具体实现代码如下所示：

```
501 struct ff_device {
502     int (*upload)(struct input_dev *dev, struct ff_effect *effect,
503                   struct ff_effect *old);
504     int (*erase)(struct input_dev *dev, int effect_id);
505
506     int (*playback)(struct input_dev *dev, int effect_id, int value);
507     void (*set_gain)(struct input_dev *dev, u16 gain);
508     void (*set_autocenter)(struct input_dev *dev, u16 magnitude);
509
510     void (*destroy)(struct ff device*);
511
512     void *private;
513
514     unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
515
516     struct mutex mutex;
517
518     int max effects;
519     struct ff_effect *effects;
520     struct file *effect_owners[];
521 };
```

13.2.2 分析核心文件input.c

文件 input.c 是输入系统驱动的核心实现，在此文件中包含了大量的操作接口。在接下来的内容中，将详细分析文件 input.c 的具体实现。

(1) 首先看函数 input_init 和 input_exit，功能是实现 input 设备的初始化和注销工作，具体实现代码如下所示：

```
2359 static int __init input_init(void)
2360 {
2361     int err;
2362
2363     err = class_register(&input_class);
2364     if (err) {
2365         pr_err("unable to register input_dev class\n");
2366         return err;
2367     }
2368
2369     err = input proc init();
2370     if (err)
2371         goto fail1;
```

```
2372
2373     err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
2374                                  INPUT_MAX_CHAR_DEVICES, "input");
2375     if (err) {
2376         pr_err("unable to register char major %d", INPUT_MAJOR);
2377         goto fail2;
2378     }
2379
2380     return 0;
2381
2382 fail2: input_proc_exit();
2383 fail1: class_unregister(&input_class);
2384     return err;
2385 }
2386
2387 static void __exit input_exit(void)
2388 {
2389     input_proc_exit();
2390     unregister_chrdev_region(MKDEV(INPUT_MAJOR, 0),
2391                              INPUT_MAX_CHAR_DEVICES);
2392     class_unregister(&input_class);
2393 }
2394
2395 subsys_initcall(input_init);
2396 module_exit(input_exit);
```

(2) 再看函数 input_allocate_device，功能是实现 input 设备的分配工作，具体实现代码如下所示：

```
1735 struct input_dev* input_allocate_device(void)
1736 {
1737     static atomic_t input_no = ATOMIC_INIT(0);
1738     struct input_dev *dev;
1739
1740     dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);
1741     if (dev) {
1742        dev->dev.type = &input_dev_type;
1743        dev->dev.class = &input_class;
1744        device_initialize(&dev->dev);
1745        mutex_init(&dev->mutex);
1746        spin_lock_init(&dev->event_lock);
1747        init_timer(&dev->timer);
1748        INIT_LIST_HEAD(&dev->h_list);
1749        INIT_LIST_HEAD(&dev->node);
1750
1751        dev_set_name(&dev->dev, "input%ld",
1752                     (unsigned long) atomic_inc_return(&input_no) - 1);
1753
1754         __module_get(THIS_MODULE);
1755     }
```

```
1756
1757     return dev;
1758 }
1759 EXPORT_SYMBOL(input_allocate_device);
```

(3) 再看函数 input_register_device，功能是实现 input 设备的注册工作，具体实现代码如下所示：

```
2025 int input_register_device(struct input_dev *dev)
2026 {
2027     struct input_devres *devres = NULL;
2028     struct input_handler *handler;
2029     unsigned int packet_size;
2030     const char *path;
2031     int error;
2032
2033     if (dev->devres_managed) {
2034         devres = devres_alloc(devm_input_device_unregister,
2035                               sizeof(struct input_devres), GFP_KERNEL);
2036         if (!devres)
2037             return -ENOMEM;
2038
2039         devres->input = dev;
2040     }
2041
2042     /* Every input device generates EV_SYN/SYN_REPORT events. */
2043     __set_bit(EV_SYN, dev->evbit);
2044
2045     /* KEY_RESERVED is not supposed to be transmitted to userspace. */
2046     __clear_bit(KEY_RESERVED, dev->keybit);
2047
2048     /* Make sure that bitmasks not mentioned in dev->evbit are clean. */
2049     input_cleanse_bitmasks(dev);
2050
2051     packet_size = input_estimate_events_per_packet(dev);
2052     if (dev->hint_events_per_packet < packet_size)
2053         dev->hint_events_per_packet = packet_size;
2054
2055     dev->max_vals = max(dev->hint_events_per_packet, packet_size) + 2;
2056     dev->vals = kcalloc(dev->max_vals, sizeof(*dev->vals), GFP_KERNEL);
2057     if (!dev->vals) {
2058         error = -ENOMEM;
2059         goto err_devres_free;
2060     }
2061
2062     /*
2063      * If delay and period are pre-set by the driver, then autorepeating
2064      * is handled by the driver itself and we don't do it in input.c.
2065      */
```

```
2066     if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
2067         dev->timer.data = (long)dev;
2068         dev->timer.function = input_repeat_key;
2069         dev->rep[REP_DELAY] = 250;
2070         dev->rep[REP_PERIOD] = 33;
2071     }
2072
2073     if (!dev->getkeycode)
2074         dev->getkeycode = input_default_getkeycode;
2075
2076     if (!dev->setkeycode)
2077         dev->setkeycode = input_default_setkeycode;
2078
2079     error = device_add(&dev->dev);
2080     if (error)
2081         goto err_free_vals;
2082
2083     path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
2084     pr_info("%s as %s\n",
2085             dev->name ? dev->name : "Unspecified device",
2086             path ? path : "N/A");
2087     kfree(path);
2088
2089     error = mutex_lock_interruptible(&input_mutex);
2090     if (error)
2091         goto err_device_del;
2092
2093     list_add_tail(&dev->node, &input_dev_list);
2094
2095     list_for_each_entry(handler, &input_handler_list, node)
2096             input_attach_handler(dev, handler);
2097
2098     input_wakeup_procfs_readers();
2099
2100     mutex_unlock(&input_mutex);
2101
2102     if (dev->devres_managed) {
2103         dev_dbg(dev->dev.parent, "%s: registering %s with devres.\n",
2104                  __func__, dev_name(&dev->dev));
2105         devres_add(dev->dev.parent, devres);
2106     }
2107     return 0;
2108
2109 err_device_del:
2110     device_del(&dev->dev);
2111 err_free_vals:
2112     kfree(dev->vals);
2113     dev->vals = NULL;
2114 err_devres_free:
```

```
2115     devres_free(devres);
2116     return error;
2117 }
2118 EXPORT_SYMBOL(input_register_device);
```

(4) 再看函数 input_unregister_device，功能是实现 input 设备的注销工作，具体实现代码如下所示：

```
2127 void input_unregister_device(struct input_dev *dev)
2128 {
2129     if (dev->devres_managed) {
2130         WARN_ON(devres_destroy(dev->dev.parent,
2131                                devm_input_device_unregister,
2132                                devm_input_device_match,
2133                                dev));
2134         __input_unregister_device(dev);
2135         /*
2136          * We do not do input_put_device() here because it will be done
2137          * when 2nd devres fires up.
2138          */
2139     } else {
2140         __input_unregister_device(dev);
2141         input_put_device(dev);
2142     }
2143 }
2144 EXPORT_SYMBOL(input_unregister_device);
```

(5) 再看函数 input_proc_init，功能是建立 input 子系统在 proc 文件系统中的目录和文件，并注册相应的 fops。函数 input_proc_init 的具体实现代码如下所示：

```
1252 static int __init input_proc_init(void)
1253 {
1254     struct proc_dir_entry *entry;
1255
1256     proc_bus_input_dir = proc_mkdir("bus/input", NULL);
1257     if (!proc_bus_input_dir)
1258         return -ENOMEM;
1259
1260     entry = proc_create("devices", 0, proc_bus_input_dir,
1261                         &input_devices_fileops);
1262     if (!entry)
1263         goto fail1;
1264
1265     entry = proc_create("handlers", 0, proc_bus_input_dir,
1266                         &input_handlers_fileops);
1267     if (!entry)
1268         goto fail2;
1269
1270     return 0;
1271
```

```
1272 fail2: remove_proc_entry("devices", proc_bus_input_dir);
1273 fail1: remove proc entry("bus/input", NULL);
1274     return -ENOMEM;
1275 }
```

(6)　再看函数 input_register_handler，功能是实现 handler 的注册，具体实现代码如下所示：

```
2154 int input_register_handler(struct input_handler *handler)
2155 {
2156     struct input_dev *dev;
2157     int error;
2158
2159     error = mutex_lock_interruptible(&input_mutex);
2160     if (error)
2161         return error;
2162
2163     INIT LIST HEAD(&handler->h list);
2164
2165     list_add_tail(&handler->node, &input_handler_list);
2166
2167     list_for_each_entry(dev, &input_dev_list, node)
2168         input_attach_handler(dev, handler);
2169
2170     input_wakeup_procfs_readers();
2171
2172     mutex_unlock(&input_mutex);
2173     return 0;
2174 }
2175 EXPORT_SYMBOL(input_register_handler);
```

(7)　再看函数 input_to_handler，功能是输入先通过所有过滤器处理后，如果没有被筛选出来，则打开所有的句柄，通过 dev->event_loc 调用实现中断禁止操作。

函数 input_to_handler 的具体实现代码如下所示：

```
 96 static unsigned int input_to_handler(struct input_handle *handle,
 97                   struct input_value *vals, unsigned int count)
 98 {
 99     struct input_handler *handler = handle->handler;
100     struct input_value *end = vals;
101     struct input_value *v;
102
103     for (v=vals; v!=vals+count; v++) {
104         if (handler->filter &&
105           handler->filter(handle, v->type, v->code, v->value))
106             continue;
107         if (end != v)
108             *end = *v;
109         end++;
110     }
111
```

```
112     count = end - vals;
113     if (!count)
114         return 0;
115
116     if (handler->events)
117         handler->events(handle, vals, count);
118     else if (handler->event)
119         for (v = vals; v != end; v++)
120             handler->event(handle, v->type, v->code, v->value);
121
122     return count;
123 }
```

(8) 再看函数 input_handle_event，功能是判断输入的 type 类型是否支持，并接着进入处理核心。函数 input_handle_event 的具体实现代码如下所示：

```
363 static void input_handle_event(struct input_dev *dev,
364   unsigned int type, unsigned int code, int value)
365 {
366     int disposition;
367
368     disposition = input_get_disposition(dev, type, code, value);
369
370     if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
371         dev->event(dev, type, code, value);
372
373     if (!dev->vals)
374         return;
375
376     if (disposition & INPUT_PASS_TO_HANDLERS) {
377         struct input_value *v;
378
379         if (disposition & INPUT_SLOT) {
380             v = &dev->vals[dev->num_vals++];
381             v->type = EV_ABS;
382             v->code = ABS_MT_SLOT;
383             v->value = dev->mt->slot;
384         }
385
386         v = &dev->vals[dev->num_vals++];
387         v->type = type;
388         v->code = code;
389         v->value = value;
390     }
391
392     if (disposition & INPUT_FLUSH) {
393         if (dev->num_vals >= 2)
394             input_pass_values(dev, dev->vals, dev->num_vals);
395         dev->num_vals = 0;
```

```
396     } else if (dev->num_vals >= dev->max_vals - 2) {
397         dev->vals[dev->num_vals++] = input_value_sync;
398         input_pass_values(dev, dev->vals, dev->num_vals);
399         dev->num_vals = 0;
400     }
401
402 }
```

(9) 再看函数 input_get_disposition，功能是获得事件处理者身份。

INPUT_PASS_TO_HANDLERS 表示交给 input handler 处理，INPUT_PASS_TO_DEVICE 表示交给 input device 处理，INPUT_FLUSH 表示需要 handler 立即处理。如果事件正常，一般返回的是 INPUT_PASS_TO_HANDLERS，只有 code 为 SYN_REPORT 时才会返回 INPUT_PASS_TO_HANDLERS | INPUT_FLUSH。

函数 input_get_disposition 的具体实现代码如下所示：

```
259 static int input_get_disposition(struct input_dev *dev,
260   unsigned int type, unsigned int code, int value)
261 {
262     int disposition = INPUT_IGNORE_EVENT;
263
264     switch (type) {
265
266     case EV_SYN:
267         switch (code) {
268             case SYN_CONFIG:
269                     disposition = INPUT_PASS_TO_ALL;
270                     break;
271
272             case SYN_REPORT:
273                     disposition = INPUT_PASS_TO_HANDLERS | INPUT_FLUSH;
274                     break;
275             case SYN_MT_REPORT:
276                     disposition = INPUT_PASS_TO_HANDLERS;
277                     break;
278         }
279         break;
280
281     case EV_KEY:
282         if (is_event_supported(code, dev->keybit, KEY_MAX)) {
283
284             /* auto-repeat bypasses state updates */
285             if (value == 2) {
286                 disposition = INPUT_PASS_TO_HANDLERS;
287                 break;
288             }
289
290             if (!!test_bit(code, dev->key) != !!value) {
291
292               __change_bit(code, dev->key);
```

```
293                 disposition = INPUT_PASS_TO_HANDLERS;
294             }
295         }
296         break;
297 
298     case EV_SW:
299         if (is_event_supported(code, dev->swbit, SW_MAX) &&
300           !!test_bit(code, dev->sw) != !!value) {
301 
302             __change_bit(code, dev->sw);
303             disposition = INPUT_PASS_TO_HANDLERS;
304         }
305         break;
306 
307     case EV_ABS:
308         if (is_event_supported(code, dev->absbit, ABS_MAX))
309             disposition = input_handle_abs_event(dev, code, &value);
310 
311         break;
312 
313     case EV REL:
314         if (is_event_supported(code, dev->relbit, REL_MAX) && value)
315             disposition = INPUT_PASS_TO_HANDLERS;
316 
317         break;
318 
319     case EV_MSC:
320         if (is_event_supported(code, dev->mscbit, MSC_MAX))
321             disposition = INPUT PASS TO ALL;
322 
323         break;
324 
325     case EV_LED:
326         if (is_event_supported(code, dev->ledbit, LED_MAX) &&
327           !!test_bit(code, dev->led) != !!value) {
328 
329              __change_bit(code, dev->led);
330              disposition = INPUT_PASS_TO_ALL;
331         }
332         break;
333 
334     case EV_SND:
335         if (is_event_supported(code, dev->sndbit, SND_MAX)) {
336 
337             if (!!test_bit(code, dev->snd) != !!value)
338                 __change_bit(code, dev->snd);
339             disposition = INPUT_PASS_TO_ALL;
340         }
341         break;
```

```
342
343     case EV_REP:
344         if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
345             dev->rep[code] = value;
346             disposition = INPUT_PASS_TO_ALL;
347         }
348         break;
349
350     case EV_FF:
351         if (value >= 0)
352             disposition = INPUT_PASS_TO_ALL;
353         break;
354
355     case EV_PWR:
356         disposition = INPUT_PASS_TO_ALL;
357         break;
358     }
359
360     return disposition;
361 }
```

(10) 再看函数 input_handle_abs_event，功能是将上次值和这次值相同的事件过滤掉。函数 input_handle_abs_event 的具体实现代码如下所示：

```
209 static int input_handle_abs_event(struct input_dev *dev,
210   unsigned int code, int *pval)
211 {
212     struct input_mt *mt = dev->mt;
213     bool is_mt_event;
214     int *pold;
215
216     if (code == ABS_MT_SLOT) {
217         /*
218          * "Stage" the event; we'll flush it later, when we
219          * get actual touch data.
220          */
221         if (mt && *pval>=0 && *pval<mt->num_slots)
222              mt->slot = *pval;
223
224         return INPUT_IGNORE_EVENT;
225     }
226
227     is_mt_event = input_is_mt_value(code);
228
229     if (!is_mt_event) {
230         pold = &dev->absinfo[code].value;
231     } else if (mt) {
232         pold = &mt->slots[mt->slot].abs[code - ABS_MT_FIRST];
233     } else {
```

```
234         /*
235          * Bypass filtering for multi-touch events when
236          * not employing slots.
237          */
238         pold = NULL;
239     }
240
241     if (pold) {
242         *pval = input_defuzz_abs_event(*pval, *pold,
243                                  dev->absinfo[code].fuzz);
244         if (*pold == *pval)
245             return INPUT_IGNORE_EVENT;
246
247         *pold = *pval;
248     }
249
250     /* Flush pending "slot" event */
251     if (is_mt_event && mt && mt->slot != input_abs_get_val(dev, ABS_MT_SLOT)) {
252         input_abs_set_val(dev, ABS_MT_SLOT, mt->slot);
253         return INPUT_PASS_TO_HANDLERS | INPUT_SLOT;
254     }
255
256     return INPUT_PASS_TO_HANDLERS;
257 }
```

在上述过滤处理过程中，如果 code 不是在 ABS_MT_FIRST 到 ABS_MT_LAST 之间，那就是单点上报(比如 ABS_X)，否则符合多点上报。上述各种情况的事件值 value 存储的位置是不一样的，所以取 pold 指针的方式是不同的(这个 pold 是过滤之后存的*pold=*pval;)。

input_defuzz_abs_event()会对比当前 value 和上一次的 old value。如果一样，就过滤掉；而不会产生任何事件。

(11) 再看函数 input_pass_values，功能是处理过滤通过的输入值，具体实现代码如下所示：

```
130 static void input_pass_values(struct input_dev *dev,
131   struct input_value *vals, unsigned int count)
132 {
133     struct input handle *handle;
134     struct input_value *v;
135
136     if (!count)
137         return;
138
139     rcu_read_lock();
140
141     handle = rcu_dereference(dev->grab);
142     if (handle) {
143         count = input_to_handler(handle, vals, count);
144     } else {
145         list_for_each_entry_rcu(handle, &dev->h_list, d_node)
146             if (handle->open)
```

```
147                 count = input_to_handler(handle, vals, count);
148         }
149 
150         rcu_read_unlock();
151 
152         add_input_randomness(vals->type, vals->code, vals->value);
153 
154         /* trigger auto repeat for key events */
155         for (v=vals; v!=vals+count; v++) {
156             if (v->type==EV_KEY && v->value!=2) {
157                 if (v->value)
158                     input_start_autorepeat(dev, v->code);
159                 else
160                     input_stop_autorepeat(dev);
161             }
162         }
163 }
```

(12) 再看函数 input_inject_event，功能是向底层发送事件，具体实现代码如下所示：

```
446 void input_inject_event(struct input_handle *handle,
447   unsigned int type, unsigned int code, int value)
448 {
449     struct input_dev *dev = handle->dev;
450     struct input_handle *grab;
451     unsigned long flags;
452 
453     if (is_event_supported(type, dev->evbit, EV_MAX)) {
454         spin_lock_irqsave(&dev->event_lock, flags);
455 
456         rcu_read_lock();
457         grab = rcu_dereference(dev->grab);
458         if (!grab || grab == handle)
459             input_handle_event(dev, type, code, value);
460         rcu_read_unlock();
461 
462         spin_unlock_irqrestore(&dev->event_lock, flags);
463     }
464 }
465 EXPORT_SYMBOL(input_inject_event);
```

(13) 再看函数 input_grab_device，这是一个 grab 抓取处理句柄函数，当输入希望处理自己的设备时，输入到处理设备中所产生的所有事件都传递这个句柄。

函数 input_grab_device 的具体实现代码如下所示：

```
512 int input_grab_device(struct input_handle *handle)
513 {
514     struct input_dev *dev = handle->dev;
515     int retval;
516 
```

```
517     retval = mutex_lock_interruptible(&dev->mutex);
518     if (retval)
519         return retval;
520
521     if (dev->grab) {
522         retval = -EBUSY;
523         goto out;
524     }
525
526     rcu_assign_pointer(dev->grab, handle);
527
528 out:
529     mutex_unlock(&dev->mutex);
530     return retval;
531 }
532 EXPORT_SYMBOL(input_grab_device);
```

(14) 再看函数 input_release_device，当一个进程 grabbed 一个设备后进行释放处理的时候调用。函数 input_release_device 的具体实现代码如下所示：

```
561 void input_release_device(struct input_handle *handle)
562 {
563     struct input_dev *dev = handle->dev;
564
565     mutex_lock(&dev->mutex);
566     __input_release_device(handle);
567     mutex_unlock(&dev->mutex);
568 }
569 EXPORT_SYMBOL(input_release_device);
```

(15) 再看函数 input_open_device，功能是打开输入设备，若 open 成功，会更新 evdev->open 计数。函数 input_open_device 的具体实现代码如下所示：

```
578 int input_open_device(struct input_handle *handle)
579 {
580     struct input_dev *dev = handle->dev;
581     int retval;
582
583     retval = mutex_lock_interruptible(&dev->mutex);
584     if (retval)
585         return retval;
586
587     if (dev->going_away) {
588         retval = -ENODEV;
589         goto out;
590     }
591
592     handle->open++;
593
594     if (!dev->users++ && dev->open)
```

```
595         retval = dev->open(dev);
596
597     if (retval) {
598         dev->users--;
599         if (!--handle->open) {
600             /*
601              * Make sure we are not delivering any more events
602              * through this handle
603              */
604             synchronize_rcu();
605         }
606     }
607
608 out:
609     mutex_unlock(&dev->mutex);
610     return retval;
611 }
612 EXPORT_SYMBOL(input_open_device);
```

(16) 再看函数 input_flush_device，功能是实现输入设备的“冲洗”处理，具体实现代码如下所示：

```
614 int input_flush_device(struct input_handle *handle, struct file *file)
615 {
616     struct input_dev *dev = handle->dev;
617     int retval;
618
619     retval = mutex_lock_interruptible(&dev->mutex);
620     if (retval)
621         return retval;
622
623     if (dev->flush)
624         retval = dev->flush(dev, file);
625
626     mutex_unlock(&dev->mutex);
627     return retval;
628 }
629 EXPORT_SYMBOL(input_flush_device);
```

(17) 再看函数 input_default_setkeycode，功能是如果没有定义有关重复按键的相关值，则用内核默认的按键值。函数 input_default_setkeycode 的具体实现代码如下所示：

```
795 static int input_default_setkeycode(struct input_dev *dev,
796    const struct input_keymap_entry *ke,
797    unsigned int *old_keycode)
798 {
799     unsigned int index;
800     int error;
801     int i;
802
```

```
803     if (!dev->keycodesize)
804         return -EINVAL;
805
806     if (ke->flags & INPUT_KEYMAP_BY_INDEX) {
807         index = ke->index;
808     } else {
809         error = input_scancode_to_scalar(ke, &index);
810         if (error)
811             return error;
812     }
813
814     if (index >= dev->keycodemax)
815         return -EINVAL;
816
817     if (dev->keycodesize < sizeof(ke->keycode) &&
818       (ke->keycode >> (dev->keycodesize * 8)))
819         return -EINVAL;
820
821     switch (dev->keycodesize) {
822         case 1: {
823             u8 *k = (u8*)dev->keycode;
824             *old_keycode = k[index];
825             k[index] = ke->keycode;
826             break;
827         }
828         case 2: {
829             u16 *k = (u16 *)dev->keycode;
830             *old_keycode = k[index];
831             k[index] = ke->keycode;
832             break;
833         }
834         default: {
835             u32 *k = (u32 *)dev->keycode;
836             *old_keycode = k[index];
837             k[index] = ke->keycode;
838             break;
839         }
840     }
841
842     __clear_bit(*old_keycode, dev->keybit);
843     __set_bit(ke->keycode, dev->keybit);
844
845     for (i=0; i<dev->keycodemax; i++) {
846         if (input_fetch_keycode(dev, i) == *old_keycode) {
847             __set_bit(*old_keycode, dev->keybit);
848             break; /* Setting the bit twice is useless, so break */
849         }
850     }
851
```

```
852     return 0;
853 }
```

(18) 再看函数 input_devices_seq_show，功能是打印输出信息到 seq 文件，接着 cat 命令会调用 read 方法，并调用 show 方法来展示输入信息。函数 input_devices_seq_show 的具体实现代码如下所示：

```
1123 static int input_devices_seq_show(struct seq_file *seq, void *v)
1124 {
1125     struct input_dev *dev = container_of(v, struct input_dev, node);
1126     const char *path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
1127     struct input_handle *handle;
1128
1129     seq_printf(seq, "I: Bus=%04x Vendor=%04x Product=%04x Version=%04x\n",
1130            dev->id.bustype, dev->id.vendor, dev->id.product, dev->id.version);
1131
1132     seq_printf(seq, "N: Name=\"%s\"\n", dev->name ? dev->name : "");
1133     seq_printf(seq, "P: Phys=%s\n", dev->phys ? dev->phys : "");
1134     seq_printf(seq, "S: Sysfs=%s\n", path ? path : "");
1135     seq_printf(seq, "U: Uniq=%s\n", dev->uniq ? dev->uniq : "");
1136     seq_printf(seq, "H: Handlers=");
1137
1138     list_for_each_entry(handle, &dev->h_list, d_node)
1139             seq_printf(seq, "%s ", handle->name);
1140     seq_putc(seq, '\n');
1141
1142     input_seq_print_bitmap(seq, "PROP", dev->propbit, INPUT_PROP_MAX);
1143
1144     input_seq_print_bitmap(seq, "EV", dev->evbit, EV_MAX);
1145     if (test_bit(EV_KEY, dev->evbit))
1146         input_seq_print_bitmap(seq, "KEY", dev->keybit, KEY_MAX);
1147     if (test_bit(EV_REL, dev->evbit))
1148         input_seq_print_bitmap(seq, "REL", dev->relbit, REL_MAX);
1149     if (test_bit(EV_ABS, dev->evbit))
1150         input_seq_print_bitmap(seq, "ABS", dev->absbit, ABS_MAX);
1151     if (test_bit(EV_MSC, dev->evbit))
1152         input_seq_print_bitmap(seq, "MSC", dev->mscbit, MSC_MAX);
1153     if (test_bit(EV_LED, dev->evbit))
1154         input_seq_print_bitmap(seq, "LED", dev->ledbit, LED_MAX);
1155     if (test_bit(EV_SND, dev->evbit))
1156         input_seq_print_bitmap(seq, "SND", dev->sndbit, SND_MAX);
1157     if (test_bit(EV_FF, dev->evbit))
1158         input_seq_print_bitmap(seq, "FF", dev->ffbit, FF_MAX);
1159     if (test_bit(EV_SW, dev->evbit))
1160         input_seq_print_bitmap(seq, "SW", dev->swbit, SW_MAX);
1161
1162     seq_putc(seq, '\n');
1163
1164     kfree(path);
```

```
1165     return 0;
1166 }
```

(19) 再看函数 input_reset_device，功能是把一个 input_dev 添加到 input_dev_list 链表上，并同时在链表 input_handler_list 中找到与这个 input_dev 相匹配的结构 input_handler，并把相匹配的 input_dev 和 input_handler 连接(connect)起来(通过 input_handle 建立连接关系)。当连接成功之后，在 input_dev 上发生的中断事件就可以传递到 input_handler 中，从而可以进一步传递到用户空间中。函数 input_reset_device 的具体实现代码如下所示：

```
1654 void input_reset_device(struct input_dev *dev)
1655 {
1656     mutex_lock(&dev->mutex);
1657
1658     if (dev->users) {
1659         input_dev_toggle(dev, true);
1660
1661         /*
1662          * Keys that have been pressed at suspend time are unlikely
1663          * to be still pressed when we resume.
1664          */
1665         spin_lock_irq(&dev->event_lock);
1666         input_dev_release_keys(dev);
1667         spin_unlock_irq(&dev->event_lock);
1668     }
1669
1670     mutex_unlock(&dev->mutex);
1671 }
1672 EXPORT_SYMBOL(input_reset_device);
```

通过本小节内容的介绍，可以总结出输入系统驱动事件的传递过程：首先在驱动层调用 input_report_abs，然后调用 input core 层的 input_event，input_event 调用了 input_handle_event 对事件进行分派，调用 input_pass_event，在这里，它会把事件传递给具体的 handler 层，然后在相应 handler 的 event 处理函数中，封装一个 event，然后把它投入 evdev 的那个 client_list 上的 client 的事件 buffer 中，等待用户空间来读取。

13.2.3 分析event机制

在 Android 系统中，输入系统 event 机制的实现文件是：

```
driver/input/event.c
```

在接下来的内容中，将详细分析 event 机制的具体实现过程。

(1) 在 Linux 内核系统中，使用结构体 input_dev 来描述一个 Input 设备，该结构的定义代码如下所示：

```
struct input_dev {
    struct input_id id; /*指向 input_id 结构*/
    bool sync;
    struct device dev; /**这些设备都归属总线设备模型*/
```

```
    struct list_head  h_list; //
    struct list_head  node;  //input_handle 链表的 list 节点
};
```

在内核中使用 input_register_device(struct input_dev *dev)来注册一个 input 设备，用 input_handler 表示 input 设备的接口，使用 input_register_handler(struct input_handler *handler)实现注册功能。

(2) 在 Event 事件驱动实现过程中，实现 Input 设备注册的代码如下所示：

```
int input_register_device(struct input_dev *dev)
{
   static atomic_t input_no = ATOMIC_INIT(0);
   struct input_handler *handler;
   const char *path;
   int error;
   /* Every input device generates EV_SYN/SYN_REPORT events. */
     set bit(EV SYN, dev->evbit); //see to inpu.h
   /* KEY_RESERVED is not supposed to be transmitted to userspace. */
   __clear_bit(KEY_RESERVED, dev->keybit);
   /* Make sure that bitmasks not mentioned in dev->evbit are clean. */
   input_cleanse_bitmasks(dev);
   /*
    * If delay and period are pre-set by the driver, then autorepeating
    * is handled by the driver itself and we don't do it in input.c.
    */
   init_timer(&dev->timer);
   //处理重复按键。如果没赋值，则为其赋默认的值
   if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
      dev->timer.data = (long)dev;
      dev->timer.function = input_repeat_key;
      dev->rep[REP_DELAY] = 250;
      dev->rep[REP_PERIOD] = 33;
   }
   if (!dev->getkeycode) //获取键的扫描码
      dev->getkeycode = input_default_getkeycode;
   if (!dev->setkeycode) //设置键值
      dev->setkeycode = input_default_setkeycode;
   dev_set_name(&dev->dev, "input%ld",
           (unsigned long) atomic_inc_return(&input_no) - 1);
   //将 input_dev 中封装的 device 注册到 sysfs
   error = device_add(&dev->dev);
   if (error)
      return error;
   path = kobject get path(&dev->dev.kobj, GFP KERNEL);
   printk(KERN_INFO "input: %s as %s\n",
      dev->name ? dev->name : "Unspecified device", path ? path : "N/A");
   kfree(path);
   error = mutex_lock_interruptible(&input_mutex);
   if (error) {
```

```
        device_del(&dev->dev);
        return error;
    }
    //将 input_device 挂到 input_dev_list 链表中
    list_add_tail(&dev->node, &input_dev_list);
    //对挂载在 input_dev_list 中的每一个 handler 调用 input_attach_handler(dev, handler);
    list_for_each_entry(handler, &input_handler_list, node)
        input_attach_handler(dev, handler);
    input_wakeup_procfs_readers();
    mutex_unlock(&input_mutex);
    return 0;
}
```

在上述代码中，首先将 input_device 挂载到 input_dev_list 链表上，然后对挂载在 input_dev_list 中的每一个 handler 调用 input_attach_handler(dev, handler)来进行匹配。例如，设备模型中的 device 和 driver 的匹配，所有的 input device 都挂载在 input_dev_list 上，所有的 handler 都挂载在 input_handler_list 上。

(3) 再看函数 input_attach_handler，功能是调用函数 input_match_device 对 handler 和 dev 通过 input_device_id *id 来进行匹配操作。如果匹配成功，则调用 handler->connect 来关联结构 input_dev *dev 和结构 input_handler *handler。

函数 input_attach_handler 的具体实现代码如下所示：

```
static int input_attach_handler(struct input_dev *dev,
  struct input_handler *handler)
{
    const struct input_device_id *id;
    int error;
    id = input_match_device(handler, dev);
    if (!id)
        return -ENODEV;
    error = handler->connect(handler, dev, id);
    if (error && error != -ENODEV)
        printk(KERN_ERR
            "input: failed to attach handler %s to device %s, ",
            "error: %d\n",
            handler->name, kobject_name(&dev->dev.kobj), error);
    return error;
}
```

函数 input_match_device 的具体实现代码如下所示：

```
static const struct input_device_id *input_match_device(
  struct input_handler *handler, struct input_dev *dev)
{
    const struct input_device_id *id;
    int i;
    for (id = handler->id_table; id->flags || id->driver_info; id++)
    { //flags 配置匹配的类型
        if (id->flags & INPUT_DEVICE_ID_MATCH_BUS) //匹配总线类型
```

```
        if (id->bustype != dev->id.bustype)
            continue;
    if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR) //匹配厂商
        if (id->vendor != dev->id.vendor)
            continue;
    if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT) //匹配制造商
        if (id->product != dev->id.product)
            continue;
    if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION) //匹配版本号
        if (id->version != dev->id.version)
            continue;
    //如果上面的 id->flags 匹配成功或者是 id->flags 没有定义，则执行下面的函数
    MATCH_BIT(evbit,  EV_MAX);
    MATCH_BIT(keybit, KEY_MAX);
    MATCH_BIT(relbit, REL_MAX);
    MATCH_BIT(absbit, ABS_MAX);
    MATCH_BIT(mscbit, MSC_MAX);
    MATCH_BIT(ledbit, LED_MAX);
    MATCH_BIT(sndbit, SND_MAX);
    MATCH_BIT(ffbit,  FF_MAX);
    MATCH_BIT(swbit,  SW_MAX);
    if (!handler->match || handler->match(handler, dev))
        return id;
  }
  return NULL;
}
```

13.3 分析硬件抽象层

经过本章 13.2 节内容的讲解，已经分析了 Android 系统中输入系统驱动的内核源码。在本节的内容中，将详细讲解输入系统硬件抽象层的实现过程，为读者步入本书后面知识的学习打下基础。

13.3.1 分析文件KeycodeLabels.h

在 Android 系统中，文件 frameworks\base\include\androidfw\KeycodeLabels.h 是本地框架层 libui 的头文件，用于实现用户空间处理功能。现实中的触摸屏和轨迹球通常非常简单，只需要传递坐标、按下、抬起等信息即可。而按键处理的过程稍微复杂，键表示方式需要先后经过按键布局转换和按键码转换。

注意：键扫描码 Scancode 是由 Linux 的输入驱动框架定义的整数类型。键扫描码 Scancode 经过一次转化后，形成按键标签 KeycodeLabel，这是一个字符串的表示形式。按键标签 KeycodeLabel 经过转换后，再次形成整数型按键码 keycode。在 Android 应用程序层，主要使用按键码 keycode 来区分。

(1) 在文件 KeycodeLabels.h 中，按键码是整数值的格式，在此文件中是使用枚举实现的，枚举 KeyCode 的定义代码如下所示：

```
typedef enum KeyCode {
    kKeyCodeUnknown = 0,
    kKeyCodeSoftLeft = 1,
    kKeyCodeSoftRight = 2,
    kKeyCodeHome = 3,
    kKeyCodeBack = 4,
    kKeyCodeCall = 5,
    kKeyCodeEndCall = 6,
    kKeyCode0 = 7,
    kKeyCode1 = 8,
    kKeyCode2 = 9,
    kKeyCode3 = 10,
    kKeyCode4 = 11,
    kKeyCode5 = 12,
    kKeyCode6 = 13,
    kKeyCode7 = 14,
    kKeyCode8 = 15,
    kKeyCode9 = 16,
    kKeyCodeStar = 17,
    kKeyCodePound = 18,
    kKeyCodeDpadUp = 19,
    kKeyCodeDpadDown = 20,
    kKeyCodeDpadLeft = 21,
    kKeyCodeDpadRight = 22,
    kKeyCodeDpadCenter = 23,
    kKeyCodeVolumeUp = 24,
    kKeyCodeVolumeDown = 25,
    kKeyCodePower = 26,
    kKeyCodeCamera = 27,
    kKeyCodeClear = 28,
    kKeyCodeA = 29,
    kKeyCodeB = 30,
    kKeyCodeC = 31,
    kKeyCodeD = 32,
    kKeyCodeE = 33,
    kKeyCodeF = 34,
    kKeyCodeG = 35,
    kKeyCodeH = 36,
    kKeyCodeI = 37,
    kKeyCodeJ = 38,
    kKeyCodeK = 39,
    kKeyCodeL = 40,
    kKeyCodeM = 41,
    kKeyCodeN = 42,
    kKeyCodeO = 43,
    kKeyCodeP = 44,
```

```
    kKeyCodeQ = 45,
    kKeyCodeR = 46,
    kKeyCodeS = 47,
    kKeyCodeT = 48,
    kKeyCodeU = 49,
    kKeyCodeV = 50,
    kKeyCodeW = 51,
    kKeyCodeX = 52,
    kKeyCodeY = 53,
    kKeyCodeZ = 54,
    kKeyCodeComma = 55,
    kKeyCodePeriod = 56,
    kKeyCodeAltLeft = 57,
    kKeyCodeAltRight = 58,
    kKeyCodeShiftLeft = 59,
    kKeyCodeShiftRight = 60,
    kKeyCodeTab = 61,
    kKeyCodeSpace = 62,
    kKeyCodeSym = 63,
    kKeyCodeExplorer = 64,
    kKeyCodeEnvelope = 65,
    kKeyCodeNewline = 66,
    kKeyCodeDel = 67,
    kKeyCodeGrave = 68,
    kKeyCodeMinus = 69,
    kKeyCodeEquals = 70,
    kKeyCodeLeftBracket = 71,
    kKeyCodeRightBracket = 72,
    kKeyCodeBackslash = 73,
    kKeyCodeSemicolon = 74,
    kKeyCodeApostrophe = 75,
    kKeyCodeSlash = 76,
    kKeyCodeAt = 77,
    kKeyCodeNum = 78,
    kKeyCodeHeadSetHook = 79,
    kKeyCodeFocus = 80,
    kKeyCodePlus = 81,
    kKeyCodeMenu = 82,
    kKeyCodeNotification = 83,
    kKeyCodeSearch = 84,
    kKeyCodePlayPause = 85,
    kKeyCodeStop = 86,
    kKeyCodeNextSong = 87,
    kKeyCodePreviousSong = 88,
    kKeyCodeRewind = 89,
    kKeyCodeForward = 90,
    kKeyCodeMute = 91
} KeyCode;
```

(2) 定义数组 KEYCODES[]，功能是存储从字符串到整数的映射关系。左列的内容表示按键标签 KeyCodeLabel，右列的内容表示按键码 KeyCode(与 KeyCode 的数值对应)。其实在按键信息第二次转化的时候，是将字符串类型 KeyCodeLabel 转化成了整数的 KeyCode。

定义数组 KEYCODES[]的代码如下所示：

```
static const KeycodeLabel KEYCODES[] = {
    { "SOFT_LEFT", 1 },
    { "SOFT_RIGHT", 2 },
    { "HOME", 3 },
    { "BACK", 4 },
    { "CALL", 5 },
    { "ENDCALL", 6 },
    { "0", 7 },
    { "1", 8 },
    { "2", 9 },
    { "3", 10 },
    { "4", 11 },
    { "5", 12 },
    { "6", 13 },
    { "7", 14 },
    { "8", 15 },
    { "9", 16 },
    { "STAR", 17 },
    { "POUND", 18 },
    { "DPAD_UP", 19 },
    { "DPAD_DOWN", 20 },
    { "DPAD_LEFT", 21 },
    { "DPAD_RIGHT", 22 },
    { "DPAD_CENTER", 23 },
    { "VOLUME_UP", 24 },
    { "VOLUME_DOWN", 25 },
    { "POWER", 26 },
    { "CAMERA", 27 },
    { "CLEAR", 28 },
    { "A", 29 },
    { "B", 30 },
    { "C", 31 },
    { "D", 32 },
    { "E", 33 },
    { "F", 34 },
    { "G", 35 },
    { "H", 36 },
    { "I", 37 },
    { "J", 38 },
    { "K", 39 },
    { "L", 40 },
    { "M", 41 },
    { "N", 42 },
```

```
    { "O", 43 },
    { "P", 44 },
    { "Q", 45 },
    { "R", 46 },
    { "S", 47 },
    { "T", 48 },
    { "U", 49 },
    { "V", 50 },
    { "W", 51 },
    { "X", 52 },
    { "Y", 53 },
    { "Z", 54 },
    { "COMMA", 55 },
    { "PERIOD", 56 },
    { "ALT_LEFT", 57 },
    { "ALT_RIGHT", 58 },
    { "SHIFT_LEFT", 59 },
    { "SHIFT_RIGHT", 60 },
    { "TAB", 61 },
    { "SPACE", 62 },
    { "SYM", 63 },
    { "EXPLORER", 64 },
    { "ENVELOPE", 65 },
    { "ENTER", 66 },
    { "DEL", 67 },
    { "GRAVE", 68 },
    { "MINUS", 69 },
    { "EQUALS", 70 },
    { "LEFT_BRACKET", 71 },
    { "RIGHT_BRACKET", 72 },
    { "BACKSLASH", 73 },
    { "SEMICOLON", 74 },
    { "APOSTROPHE", 75 },
    { "SLASH", 76 },
    { "AT", 77 },
    { "NUM", 78 },
    { "HEADSETHOOK", 79 },
    { "FOCUS", 80 },
    { "PLUS", 81 },
    { "MENU", 82 },
    { "NOTIFICATION", 83 },
    { "SEARCH", 84 },
    { "MEDIA_PLAY_PAUSE", 85 },
    { "MEDIA_STOP", 86 },
    { "MEDIA_NEXT", 87 },
    { "MEDIA_PREVIOUS", 88 },
    { "MEDIA_REWIND", 89 },
    { "MEDIA_FAST_FORWARD", 90 },
    { "MUTE", 91 },
```

```
    { NULL, 0 }
};
```

注意： 文件 frameworks/base/core/Java/android/view/KeyEvent.Java 中定义了 android.view.KeyEvent 类，在里面定义的整数类型的数值与 KeycodeLabels.h 中定义的枚举 KeyCode 值是对应的。

13.3.2 分析文件KeyCharacterMap.h

在 Android 系统中，文件 frameworks\base\include\androidfw\KeyCharacterMap.h 也是本地框架层 libui 的头文件，在里面定义了按键的字符映射关系。其实，KeyCharacterMap 只是一个辅助的功能，因为按键码只是一个与 UI 无关整数，通常用程序对其进行捕获处理，然而，如果将按键事件转换为用户可见的内容，就需要经过这个层次的转换。

KeyCharacterMap 类的有关代码如下所示：

```
class KeyCharacterMap : public RefBase {
public:
    enum KeyboardType {
        KEYBOARD_TYPE_UNKNOWN = 0,
        KEYBOARD_TYPE_NUMERIC = 1,
        KEYBOARD_TYPE_PREDICTIVE = 2,
        KEYBOARD_TYPE_ALPHA = 3,
        KEYBOARD_TYPE_FULL = 4,
        KEYBOARD_TYPE_SPECIAL_FUNCTION = 5,
        KEYBOARD_TYPE_OVERLAY = 6,
    };

    enum Format {
        // Base keyboard layout, may contain device-specific options,
        // such as "type" declaration.
        FORMAT_BASE = 0,
        // Overlay keyboard layout, more restrictive, may be published by applications,
        // cannot override device-specific options.
        FORMAT_OVERLAY = 1,
        // Either base or overlay layout ok.
        FORMAT_ANY = 2,
    };

    // Substitute key code and meta state for fallback action.
    struct FallbackAction {
        int32_t keyCode;
        int32_t metaState;
    };
```

在上述代码中，使用 KeyCharacterMap 将按键码映射为文本可识别的字符串。

注意： KeyCharacterMap 需要从本地层传送到 Java 层，其中涉及的 JNI 的代码路径如下：

```
frameworks/base/core/jni/android_text_KeyCharacterMap.cpp
```

KeyCharacterMap Java 框架层的代码如下：

```
frameworks/base/core/Java/android/view/KeyCharacterMap.Java
```

类 android.view.KeyCharacterMap 是 Android 平台的 API，我们可以在应用程序中使用这个类。另外，在 android.text.method 中有各种 Listener，相互之间可以监听 KeyCharacterMap 相关的信息。

上面关于按键码和按键字符映射的内容是在代码中实现的内容，我们还需要配合动态的配置文件来使用。在实现 Android 系统的时候，很可能需要更改这两种文件。我们需要动态配置如下两个文件。

- KL(Keycode Layout)：后缀名为 kl 的配置文件。
- KCM(KeyCharacterMap)：后缀名为 kcm 的配置文件。

在 Donut 及其之前版本的配置文件路径为：

```
development/emulator/keymaps/
```

在 Eclair 及其之后版本的配置文件路径为：

```
sdk/emulator/keymaps/
```

当系统生成上述配置文件后，将被放置在目标文件系统的/system/usr/keylayout/目录中或/system/usr/keychars/目录中。另外，kl 文件将被直接复制到目标文件系统中；由于尺寸较大，kcm 文件放置在目标文件系统中之前，需要经过压缩处理。KeyLayoutMap.cpp 负责解析处理 kl 文件，KeyCharacterMap.cpp 负责解析 kcm 文件。

13.3.3 分析Kl格式的文件

在 Android 系统中，Kl 格式的文件是按键布局文件，通常以原始的文本文件的形式存在，被保存在目标文件系统的/system/usr/keylayout/目录中或者/system/usr/keychars/目录中。

Android 默认提供的按键布局文件有两个，分别是 qwerty.kl 和 AVRCP.kl。其中 qwerty.kl 是全键盘的布局文件，是系统中主要按键使用的布局文件；文件 AVRCP.kl 用于实现多媒体的控制。

文件 qwerty.kl 的主要代码如下所示：

```
key 399   GRAVE
key 2     1
key 3     2
key 4     3
key 5     4
key 6     5
key 7     6
key 8     7
key 9     8
key 10    9
```

```
key 11    0
key 158   BACK           WAKE DROPPED
key 230   SOFT_RIGHT      WAKE
key 60    SOFT_RIGHT      WAKE
key 107   ENDCALL         WAKE_DROPPED
key 62    ENDCALL         WAKE_DROPPED
key 229   MENU           WAKE_DROPPED

# 省略部分按键的对应内容
key 16    Q
key 17    W
key 18    E
key 19    R
key 20    T
key 115   VOLUME_UP
key 114   VOLUME_DOWN
```

在上述代码中，第 1 列为按键的扫描码，是一个整数值；第 2 列为按键的标签，是一个字符串。即完成了按键信息的第 1 次转化，将整型的扫描码转换成字符串类型的按键标签。第 3 列表示按键的 Flag，带有 WAKE 字符，表示此按键可以唤醒系统。

注意： 扫描码受驱动程序决定，不同的扫描码对应一个按键标签。两个手机的物理按键可以对应同一个功能按键。假如当上面的扫描码为 158 时对应的标签为 BACK，经过第二次转换后，根据 KeycodeLabels.h 的 KEYCODES 数组可得出其对应的按键码是 4。

13.3.4　分析kcm格式文件

在 Android 系统中，kcm 格式文件是按键字符映射文件，用于表示按键字符的映射关系，功能是将整数类型按键码(keycode)转化成可以显示的字符。kcm 文件将被 makekcharmap 工具转化成二进制的格式，放在目标系统的/system/usr/keychars/目录中。

文件 qwerty.kcm 表示全键盘的字符映射关系，其主要代码如下所示：

```
[type=QWERTY]
# keycode  display number  base   caps   fn     caps fn
A              'A'    '2'    'a'    'A'    '#'    0x00
B              'B'    '2'    'b'    'B'    '<'    0x00
C              'C'    '2'    'c'    'C'    '9'    0x00E7
D              'D'    '3'    'd'    'D'    '5'    0x00
E              'E'    '3'    'e'    'E'    '2'    0x0301
F              'F'    '3'    'f'    'F'    '6'    0x00A5
G              'G'    '4'    'g'    'G'    '-'    '_'
H              'H'    '4'    'h'    'H'    '['    '{'
I              'I'    '4'    'i'    'I'    '$'    0x0302
J              'J'    '5'    'j'    'J'    ']'    '}'
K              'K'    '5'    'k'    'K'    '"'    '~'
L              'L'    '5'    'l'    'L'    '''    '`'
```

```
M              'M'    '6'    'm'    'M'    '!'    0x00
N              'N'    '6'    'n'    'N'    '>'    0x0303
```

在上述代码中，第一列表示转换之前的按键码，第二列后面的分别表示转换成的显示内容(display)和数字(number)等内容。这些转化的内容与文件 KeyCharacterMap.h 相对应，具体内容是在此文件的 getDisplayLabel()和 getNumber()等函数中定义的。

除了 QWERTY 映射类型之外，还可以映射 Q14(单键多字符对应的键盘)和 NUMERIC(12 键的数字键盘)。

13.3.5 分析文件EventHub.cpp

在 Android 系统中，文件 frameworks\base\services\input\EventHub.cpp 是输入系统的核心控制文件，整个输入系统的主要的功能都是在此文件中实现的。例如，当按下电源键后，系统把 scanCode 写入对应的设备节点，文件 EventHub.cpp 会去读这个设备节点，并把 scanCode 通过 Kl 文件对应成 keyCode 发送到上层。

在文件 EventHub.cpp 中需要定义设备节点所在的路径，定义代码如下所示：

```
static const char *WAKE_LOCK_ID = "KeyEvents";
static const char *DEVICE_PATH = "/dev/input";  //输入设备的目录
```

在具体处理时，在函数 openPlatformInput 中通过调用函数 scan_dir 搜索路径下面所有 Input 驱动的设备节点。函数 scan_dir()会从目录中查找设备，找到后，调用 open_device()函数以打开查找到的设备。其中函数 openPlatformInput()的实现代码如下所示：

```
bool EventHub::openPlatformInput(void)
{
   /*
    * Open platform-specific input device(s).
    */
   int res;

   mFDCount = 1;
   mFDs = (pollfd*)calloc(1, sizeof(mFDs[0]));
   mDevices = (device_t**)calloc(1, sizeof(mDevices[0]));
   mFDs[0].events = POLLIN;
   mDevices[0] = NULL;
#ifdef HAVE_INOTIFY
   mFDs[0].fd = inotify_init();
   res = inotify_add_watch(mFDs[0].fd, device_path, IN_DELETE | IN_CREATE);
   if(res < 0) {
      LOGE("could not add watch for %s, %s\n", device_path, strerror(errno));
   }
#else
   mFDs[0].fd = -1;
#endif

   res = scan_dir(device_path);
```

```
    if(res < 0) {
        LOGE("scan dir failed for %s\n", device path);
        //open_device("/dev/input/event0");
    }
    return true;
}
```

再看函数 getEvent()，功能是在一个无限循环之内调用阻塞的函数等待事件到来。具体实现代码如下所示：

```
bool EventHub::getEvent(int32_t *outDeviceId, int32_t *outType,
        int32_t *outScancode, int32_t *outKeycode, uint32_t *outFlags,
        int32_t *outValue, nsecs_t *outWhen)
{
    *outDeviceId = 0;
    *outType = 0;
    *outScancode = 0;
    *outKeycode = 0;
    *outFlags = 0;
    *outValue = 0;
    *outWhen = 0;

    status_t err;

    fd_set readfds;
    int maxFd = -1;
    int cc;
    int i;
    int res;
    int pollres;
    struct input_event iev;

    // Note that we only allow one caller to getEvent(), so don't need
    // to do locking here...  only when adding/removing devices.

    while(1) {

        // First, report any devices that had last been added/removed.
        if (mClosingDevices != NULL) {
            device_t *device = mClosingDevices;
            LOGV("Reporting device closed: id=0x%x, name=%s\n",
                device->id, device->path.string());
            mClosingDevices = device->next;
            *outDeviceId = device->id;
            if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
            *outType = DEVICE_REMOVED;
            delete device;
            return true;
        }
```

```
        if (mOpeningDevices != NULL) {
            device_t *device = mOpeningDevices;
            LOGV("Reporting device opened: id=0x%x, name=%s\n",
                 device->id, device->path.string());
            mOpeningDevices = device->next;
            *outDeviceId = device->id;
            if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
            *outType = DEVICE_ADDED;
            return true;
        }

        release_wake_lock(WAKE_LOCK_ID);

        pollres = poll(mFDs, mFDCount, -1);

        acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_ID);

        if (pollres <= 0) {
            if (errno != EINTR) {
                LOGW("select failed (errno=%d)\n", errno);
                usleep(100000);
            }
            continue;
        }

        //printf("poll %d, returned %d\n", mFDCount, pollres);
        if(mFDs[0].revents & POLLIN) {
            read_notify(mFDs[0].fd);
        }
        for(i=1; i<mFDCount; i++) {
            if(mFDs[i].revents) {
                LOGV("revents for %d = 0x%08x", i, mFDs[i].revents);
                if(mFDs[i].revents & POLLIN) {
                    res = read(mFDs[i].fd, &iev, sizeof(iev));
                    if (res == sizeof(iev)) {
                        LOGV("%s got: t0=%d, t1=%d, type=%d, code=%d, v=%d",
                             mDevices[i]->path.string(),
                             (int)iev.time.tv_sec, (int)iev.time.tv_usec,
                             iev.type, iev.code, iev.value);
                        *outDeviceId = mDevices[i]->id;
                        if (*outDeviceId == mFirstKeyboardId) *outDeviceId = 0;
                        *outType = iev.type;
                        *outScancode = iev.code;
                        if (iev.type == EV_KEY) {
                            err = mDevices[i]->layoutMap->map(
                              iev.code, outKeycode, outFlags);
                            LOGV("iev.code=%d outKeycode=%d outFlags=0x%08x err=%d\n",
                                iev.code, *outKeycode, *outFlags, err);
                            if (err != 0) {
```

```
                        *outKeycode = 0;
                        *outFlags = 0;
                     }
                  } else {
                     *outKeycode = iev.code;
                  }
                  *outValue = iev.value;
                  *outWhen = s2ns(iev.time.tv_sec) + us2ns(iev.time.tv_usec);
                  return true;
               } else {
                  if (res<0) {
                     LOGW("could not get event (errno=%d)", errno);
                  } else {
                     LOGE("could not get event (wrong size: %d)", res);
                  }
                  continue;
               }
            }
         }
      }
   }
}
```

上述代码中，通过函数 poll 来阻塞程序的运行，此时为等待状态，不会有内存开销。

当 Input 设备的相应事件发生后，会将函数 poll 返回，然后通过函数 read 读取 Input 设备发生的事件代码。

注意： 在 Android 系统中，会有一些 input 设备可能不需要经过 EventHub 处理，在这种情况下，可以根据 EventHub 中的 open_device()函数进行处理。我们可以在驱动程序中设置一些标志来屏蔽一些设备。在函数 open_device()中实现了键盘、轨迹球和触摸屏等几种设备的处理功能，对其他设备可以忽略。此外还有另外一种简单的方法实现设备忽略功能，即把不需要 EventHub 处理的设备节点不放置在/dev/input 目录中。

另外，函数 open_device 还可以打开并处理 system/usr/keylayout/目录中的 Kl 文件，此函数对应此功能的实现代码如下所示：

```
const char *root = getenv("ANDROID_ROOT");
snprintf(keylayoutFilename, sizeof(keylayoutFilename),
         "%s/usr/keylayout/%s.kl", root, tmpfn);
bool defaultKeymap = false;
if (access(keylayoutFilename, R_OK)) {
    snprintf(keylayoutFilename, sizeof(keylayoutFilename),
             "%s/usr/keylayout/%s", root, "qwerty.kl");
    defaultKeymap = true;
}
```

在 Android 中已经定义了丰富、完整的标准按键。在一般情况下，我们只需要根据 Kl 配

置按键即可，不需要再为 Android 系统增加按键。当在现实项目中需要较为奇特的按键的时候，我们需要更改 Android 系统的框架层，来实现更改按键功能。

在 Android 中增加新按键时，需要更改下面的文件。

- KeycodeLabels.h：保存在 frameworks/base/include/ui/目录下，需要修改 KeyCode 枚举数值和 KeycodeLabel 类型 Code 数组。
- KeyEvent.Java：保存在 frameworks/base/core/Java/android/view/目录下，在此可以定义整数值作为平台的 API，供 Java 应用程序使用。
- attrs.xml：保存在 frameworks/base/core/res/res/values/目录下，表示属性的资源文件，需要修改其中的 name="keycode"的 attr。框架层增加完成后，只需要更改 Kl 文件，增加按键的映射关系即可。

除此之外，还有一种更为简易的做法，就是使用 Android 中已经定义的“特殊”按键码作为这个新增按键的键码。使用这种方式，Android 的框架层不需要做任何改动。这种方式的潜在问题是，当某些第三方的应用可能已经使用那些特殊按键时，会意外激发系统的这种新增的按键。

13.4　分析驱动的具体实现

经过本章前面内容的讲解，已经了解了 Android 输入系统的内核和硬件抽象层的具体实现过程。在本节的内容中，将依次讲解 Android 内置的模拟器、MSM 内核和 OMAP 内核中输入驱动的具体实现流程。

13.4.1　分析内置模拟器中的输入驱动实现

在 GoldFish 虚拟处理器中，使用 event 驱动程序作为键盘输入功能的驱动程序，其驱动程序的相关文件是 drivers/input/keyboard/goldfish_events.c。此驱动程序是一个标准的 event 驱动程序，在用户空间的设备节点为/dev/event/event0，其核心代码如下所示：

```
static irqreturn_t events_interrupt(int irq, void *dev_id)
{
   struct event_dev *edev = dev_id;
   unsigned type, code, value;
   type = __raw_readl(edev->addr + REG_READ);     // 类型
   code = __raw_readl(edev->addr + REG_READ);     // 码
   value = __raw_readl(edev->addr + REG_READ);    // 数值
   input_event(edev->input, type, code, value);
   return IRQ_HANDLED;
}
```

函数 events_interrupt 是按键事件的中断处理函数，当中断发生后，会读取虚拟寄存器的内容，并将信息上报。模拟器根据主机环境键盘按下的情况，可以得到虚拟寄存器中的内容。

在模拟器环境中，使用默认的所有的 Kl 和 KCM 文件，由于模拟器环境支持全键盘，因此基本上包含了大部分的功能。在模拟器环境中，实际上按键的扫描码对应的是桌面电脑的键

盘(效果与鼠标点击模拟器的控制面板类似)。当按下键盘的某些按键后，会转化为驱动程序中的扫描码，然后再由上层的用户空间处理。上述过程与实际系统是类似的。通过更改默认 Kl 文件的方式，又可以更改实际按键的映射关系。

13.4.2 MSM高通处理器中的输入驱动实现

在高通 MSM 的 Mahimahi 平台中，具有触摸屏、轨迹球和简单按键功能。这些功能是在 Android 系统中由驱动程序实现的，并且需要用户空间的内容来协助实现。

在 Mahimahi 平台中，输入系统设备包括了以下 Event 设备。

- /dev/input/event4：几个按键的设备。
- /dev/input/event2：触摸屏设备。
- /dev/input/event5：轨迹球设备。

1. 触摸屏驱动

高通 Mahimahi 平台的触摸屏驱动程序的实现文件是 drivers/input/touchscreen/synaptics_i2c_rmi.c，此文件的核心是 synaptics_ts_probe()函数，在该函数中，需要进行触摸屏工作模式的初始化，对作为输入设备的触摸屏驱动在 Linux 平台下进行设备名注册，同时初始化触摸事件触发时引起的中断操作。此函数的实现代码如下所示：

```
static int synaptics_ts_probe(
  struct i2c_client *client, const struct i2c_device_id *id)
{
    struct synaptics_ts_data *ts;
    uint8_t buf0[4];
    uint8_t buf1[8];
    struct i2c_msg msg[2];
    int ret = 0;
    uint16_t max_x, max_y;
    int fuzz_x, fuzz_y, fuzz_p, fuzz_w;
    struct synaptics_i2c_rmi_platform_data *pdata;
    int inactive_area_left;
    int inactive_area_right;
    int inactive_area_top;
    int inactive_area_bottom;
    int snap_left_on;
    int snap_left_off;
    int snap_right_on;
    int snap_right_off;
    int snap_top_on;
    int snap_top_off;
    int snap_bottom_on;
    int snap_bottom_off;
    uint32_t panel_version;

    if (!i2c_check_functionality(client->adapter, I2C_FUNC_I2C)) {
        printk(KERN_ERR "synaptics_ts_probe: need I2C_FUNC_I2C\n");
```

```
        ret = -ENODEV;
        goto err_check_functionality_failed;
    }

    ts = kzalloc(sizeof(*ts), GFP_KERNEL);
    if (ts == NULL) {
        ret = -ENOMEM;
        goto err_alloc_data_failed;
    }
    INIT_WORK(&ts->work, synaptics_ts_work_func);
    ts->client = client;
    i2c_set_clientdata(client, ts);
    pdata = client->dev.platform_data;
    if (pdata)
        ts->power = pdata->power;
    if (ts->power) {
        ret = ts->power(1);
        if (ret < 0) {
            printk(KERN_ERR "synaptics_ts_probe power on failed\n");
            goto err_power_failed;
        }
    }

    ret = i2c_smbus_write_byte_data(ts->client, 0xf4, 0x01); /* device command = reset */
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed\n");
        /* fail? */
    }
    {
        int retry = 10;
        while (retry-- > 0) {
            ret = i2c_smbus_read_byte_data(ts->client, 0xe4);
            if (ret >= 0)
                break;
            msleep(100);
        }
    }
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: Product Major Version %x\n", ret);
    panel_version = ret << 8;
    ret = i2c_smbus_read_byte_data(ts->client, 0xe5);
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: Product Minor Version %x\n", ret);
```

```
    panel_version |= ret;

    ret = i2c_smbus_read_byte_data(ts->client, 0xe3);
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: product property %x\n", ret);

    if (pdata) {
        while (pdata->version > panel_version)
            pdata++;
        ts->flags = pdata->flags;
        inactive_area_left = pdata->inactive_left;
        inactive_area_right = pdata->inactive_right;
        inactive_area_top = pdata->inactive_top;
        inactive_area_bottom = pdata->inactive_bottom;
        snap_left_on = pdata->snap_left_on;
        snap_left_off = pdata->snap_left_off;
        snap_right_on = pdata->snap_right_on;
        snap_right_off = pdata->snap_right_off;
        snap_top_on = pdata->snap_top_on;
        snap_top_off = pdata->snap_top_off;
        snap_bottom_on = pdata->snap_bottom_on;
        snap_bottom_off = pdata->snap_bottom_off;
        fuzz_x = pdata->fuzz_x;
        fuzz_y = pdata->fuzz_y;
        fuzz_p = pdata->fuzz_p;
        fuzz_w = pdata->fuzz_w;
    } else {
        inactive_area_left = 0;
        inactive_area_right = 0;
        inactive_area_top = 0;
        inactive_area_bottom = 0;
        snap_left_on = 0;
        snap_left_off = 0;
        snap_right_on = 0;
        snap_right_off = 0;
        snap_top_on = 0;
        snap_top_off = 0;
        snap_bottom_on = 0;
        snap_bottom_off = 0;
        fuzz_x = 0;
        fuzz_y = 0;
        fuzz_p = 0;
        fuzz_w = 0;
    }

    ret = i2c_smbus_read_byte_data(ts->client, 0xf0);
```

```
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: device control %x\n", ret);

    ret = i2c_smbus_read_byte_data(ts->client, 0xf1);
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_byte_data failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: interrupt enable %x\n", ret);

    ret = i2c_smbus_write_byte_data(ts->client, 0xf1, 0); /* disable interrupt */
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed\n");
        goto err_detect_failed;
    }

    msg[0].addr = ts->client->addr;
    msg[0].flags = 0;
    msg[0].len = 1;
    msg[0].buf = buf0;
    buf0[0] = 0xe0;
    msg[1].addr = ts->client->addr;
    msg[1].flags = I2C_M_RD;
    msg[1].len = 8;
    msg[1].buf = buf1;
    ret = i2c_transfer(ts->client->adapter, msg, 2);
    if (ret < 0) {
        printk(KERN_ERR "i2c_transfer failed\n");
        goto err_detect_failed;
    }
    printk(KERN_INFO "synaptics_ts_probe: 0xe0: %x %x %x %x %x %x %x %x\n",
           buf1[0], buf1[1], buf1[2], buf1[3],
             buf1[4], buf1[5], buf1[6], buf1[7]);

    ret = i2c_smbus_write_byte_data(ts->client, 0xff, 0x10); /* page select = 0x10 */
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_write_byte_data failed for page select\n");
        goto err_detect_failed;
    }
    ret = i2c_smbus_read_word_data(ts->client, 0x04);
    if (ret < 0) {
        printk(KERN_ERR "i2c_smbus_read_word_data failed\n");
        goto err_detect_failed;
    }
    ts->max[0] = max_x = (ret >> 8 & 0xff) | ((ret & 0x1f) << 8);
    ret = i2c_smbus_read_word_data(ts->client, 0x06);
```

```
    if (ret < 0) {
        printk(KERN ERR "i2c smbus read word data failed\n");
        goto err_detect_failed;
    }
    ts->max[1] = max_y = (ret >> 8 & 0xff) | ((ret & 0x1f) << 8);
    if (ts->flags & SYNAPTICS_SWAP_XY)
        swap(max_x, max_y);

    ret = synaptics_init_panel(ts); /* will also switch back to page 0x04 */
    if (ret < 0) {
        printk(KERN_ERR "synaptics_init_panel failed\n");
        goto err_detect_failed;
    }

    ts->input_dev = input_allocate_device();  //创建设备
    if (ts->input_dev == NULL) {
        ret = -ENOMEM;
        printk(KERN_ERR "synaptics_ts_probe: Failed to allocate input device\n");
        goto err_input_dev_alloc_failed;
    }
    ts->input dev->name = "synaptics-rmi-touchscreen";
    //声明输入设备
    set_bit(EV_SYN, ts->input_dev->evbit);
    set_bit(EV_KEY, ts->input_dev->evbit);
    set_bit(BTN_TOUCH, ts->input_dev->keybit);
    set_bit(BTN_2, ts->input_dev->keybit);
    set_bit(EV_ABS, ts->input_dev->evbit);
    inactive_area_left = inactive_area_left * max_x / 0x10000;
    inactive area right = inactive area right * max x / 0x10000;
    inactive_area_top = inactive_area_top * max_y / 0x10000;
    inactive_area_bottom = inactive_area_bottom * max_y / 0x10000;
    snap_left_on = snap_left_on * max_x / 0x10000;
    snap_left_off = snap_left_off * max_x / 0x10000;
    snap_right_on = snap_right_on * max_x / 0x10000;
    snap_right_off = snap_right_off * max_x / 0x10000;
    snap_top_on = snap_top_on * max_y / 0x10000;
    snap_top_off = snap_top_off * max_y / 0x10000;
    snap_bottom_on = snap_bottom_on * max_y / 0x10000;
    snap_bottom_off = snap_bottom_off * max_y / 0x10000;
    fuzz_x = fuzz_x * max_x / 0x10000;
    fuzz_y = fuzz_y * max_y / 0x10000;
    ts->snap_down[!!(ts->flags & SYNAPTICS_SWAP_XY)] = -inactive_area_left;
    ts->snap_up[!!(ts->flags & SYNAPTICS_SWAP_XY)] = max_x + inactive_area_right;
    ts->snap_down[!(ts->flags & SYNAPTICS_SWAP_XY)] = -inactive_area_top;
    ts->snap_up[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y + inactive_area_bottom;
    ts->snap_down_on[!!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_left_on;
    ts->snap_down_off[!!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_left_off;
    ts->snap up on[!!(ts->flags & SYNAPTICS SWAP XY)] = max x - snap right on;
    ts->snap_up_off[!!(ts->flags & SYNAPTICS_SWAP_XY)] = max_x - snap_right_off;
```

```
    ts->snap_down_on[!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_top_on;
    ts->snap_down_off[!(ts->flags & SYNAPTICS_SWAP_XY)] = snap_top_off;
    ts->snap_up_on[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y - snap_bottom_on;
    ts->snap_up_off[!(ts->flags & SYNAPTICS_SWAP_XY)] = max_y - snap_bottom_off;
    printk(KERN_INFO "synaptics_ts_probe: max_x %d, max_y %d\n", max_x, max_y);
    printk(KERN_INFO "synaptics_ts_probe: inactive_x %d %d, inactive_y %d %d\n",
           inactive_area_left, inactive_area_right,
           inactive_area_top, inactive_area_bottom);
    printk(KERN_INFO "synaptics_ts_probe: snap_x %d-%d %d-%d, snap_y %d-%d %d-%d\n",
           snap_left_on, snap_left_off, snap_right_on, snap_right_off,
           snap_top_on, snap_top_off, snap_bottom_on, snap_bottom_off);
    //配置具体事件
    input_set_abs_params(ts->input_dev, ABS_X, -inactive_area_left,
      max_x + inactive_area_right, fuzz_x, 0);
    input_set_abs_params(ts->input_dev, ABS_Y, -inactive_area_top,
      max_y + inactive_area_bottom, fuzz_y, 0);
    input_set_abs_params(ts->input_dev, ABS_PRESSURE, 0, 255, fuzz_p, 0);
    input_set_abs_params(ts->input_dev, ABS_TOOL_WIDTH, 0, 15, fuzz_w, 0);
    input_set_abs_params(ts->input_dev, ABS_HAT0X, -inactive_area_left,
      max_x + inactive_area_right, fuzz_x, 0);
    input_set_abs_params(ts->input_dev, ABS_HAT0Y, -inactive_area_top,
      max_y + inactive_area_bottom, fuzz_y, 0);
    /* ts->input_dev->name = ts->keypad_info->name; */
    ret = input_register_device(ts->input_dev);
    if (ret) {
        printk(KERN_ERR "synaptics_ts_probe: Unable to register %s input device\n",
                ts->input_dev->name);
        goto err_input_register_device_failed;
    }
    if (client->irq) {
        ret = request_irq(client->irq, synaptics_ts_irq_handler, 0, client->name, ts);
        if (ret == 0) {
            ret = i2c_smbus_write_byte_data(ts->client, 0xf1, 0x01); /* enable abs int */
            if (ret)
                free_irq(client->irq, ts);
        }
        if (ret == 0)
            ts->use_irq = 1;
        else
            dev_err(&client->dev, "request_irq failed\n");
    }
    if (!ts->use_irq) {
        hrtimer_init(&ts->timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        ts->timer.function = synaptics_ts_timer_func;
        hrtimer_start(&ts->timer, ktime_set(1, 0), HRTIMER_MODE_REL);
    }
#ifdef CONFIG_HAS_EARLYSUSPEND
    ts->early_suspend.level = EARLY_SUSPEND_LEVEL_BLANK_SCREEN + 1;
```

```
    ts->early_suspend.suspend = synaptics_ts_early_suspend;
    ts->early_suspend.resume = synaptics_ts_late_resume;
    register_early_suspend(&ts->early_suspend);
#endif

    printk(KERN_INFO "synaptics_ts_probe: Start touchscreen %s in %s mode\n",
      ts->input_dev->name, ts->use_irq ? "interrupt" : "polling");

    return 0;

err_input_register_device_failed:
    input_free_device(ts->input_dev);

err_input_dev_alloc_failed:
err_detect_failed:
err_power_failed:
    kfree(ts);
err_alloc_data_failed:
err_check_functionality_failed:
    return ret;
}
```

在上述代码中，通过 i2c_smbus_read_byte_data()函数对寄存器信息进行读取即可完成事件信息的获取，也可以通过 i2c_transfer 完成对寄存器信息的批量读取。

2. 按键和轨迹球驱动

MSM 具有按键和轨迹球的功能，对应的驱动程序在文件 arch/arm/mach-msm/board-mahimahi-keypad.c 中，接下来开始介绍此文件的实现流程。

(1) 文件 board-mahimahi-keypad.c 中的全局定义代码如下所示：

```
static struct gpio_event_info *mahimahi_input_info[] = {
   &mahimahi_keypad_matrix_info.info,      // 键盘矩阵
   &mahimahi_keypad_key_info.info,        // 键盘信息
   &jogball_x_axis.info.info,            // 轨迹球 X 方向信息
   &jogball_y_axis.info.info,            // 轨迹球 Y 方向信息
};
static struct gpio_event_platform_data mahimahi_input_data = {
   .names = {
      "mahimahi-keypad",                // 按键设备
      "mahimahi-nav",                  // 轨迹球设备
      NULL,
   },
   .info = mahimahi_input_info,
   .info_count = ARRAY_SIZE(mahimahi_input_info),
   .power = jogball_power,
};
static struct platform_device mahimahi_input_device = {
```

```
    .name = GPIO_EVENT_DEV_NAME,
    .id = 0,
    .dev = {
        .platform_data = &mahimahi_input_data,
    },
};
```

因为按键和轨迹球是通过 GPIO 系统实现的，所以在上面定义了一个 gpio_event_info 类型的数组。mahimahi-keypad 和 mahimahi-nav 分别表示两个设备的名称。在 gpio_event_info 指针数组 mahimahi_input_info 中包括 4 个成员，分别是 mahimahi_keypad_matrix_info.info、mahimahi_keypad_key_ info.info、jogball_x_axis.info.info 和 jogball_y_axis.info.info。

(2)　使用 gpio_event_matrix_info 矩阵定义按键驱动，此驱动是利用 GPIO 矩阵实现的，在定义时，需要包含按键的 GPIO 矩阵和 input 设备的信息，具体代码如下所示：

```
static unsigned int mahimahi_col_gpios[] = { 33, 32, 31 };
static unsigned int mahimahi_row_gpios[] = { 42, 41, 40 };

#define KEYMAP_INDEX(col, row)  ((col)*ARRAY_SIZE(mahimahi_row_gpios) + (row))
#define KEYMAP_SIZE  (ARRAY_SIZE(mahimahi_col_gpios) * \
                 ARRAY_SIZE(mahimahi_row_gpios))
static const unsigned short mahimahi_keymap
[KEYMAP_SIZE] = {        // 按键映射关系
    [KEYMAP_INDEX(0, 0)] = KEY_VOLUMEUP, /* 115 */
    [KEYMAP_INDEX(0, 1)] = KEY_VOLUMEDOWN, /* 114 */
    [KEYMAP_INDEX(1, 1)] = MATRIX_KEY(1, BTN_MOUSE),
};
static struct gpio_event_matrix_info mahimahi_keypad_matrix_info = {
    .info.func = gpio_event_matrix_func,
    // 关键函数的实现
    .keymap = mahimahi_keymap,
    .output_gpios = mahimahi_col_gpios,
    .input_gpios = mahimahi_row_gpios,
    .noutputs = ARRAY_SIZE(mahimahi_col_gpios),
    .ninputs = ARRAY_SIZE(mahimahi_row_gpios),
    .settle_time.tv.nsec = 40 * NSEC_PER_USEC,
    .poll_time.tv.nsec = 20 * NSEC_PER_MSEC,
    .flags = (GPIOKPF_LEVEL_TRIGGERED_IRQ
        | GPIOKPF_REMOVE_PHANTOM_KEYS
        | GPIOKPF_PRINT_UNMAPPED_KEYS),
};
static struct gpio_event_direct_entry mahimahi_keypad_key_map[] = {//Power 按键
    {
        .gpio   = MAHIMAHI_GPIO_POWER_KEY,
        .code   = KEY_POWER,
    },
};
static struct gpio_event_input_info mahimahi_keypad_key_info = {
    .info.func = gpio_event_input_func,
```

```
    // 关键函数实现
    .info.no suspend = true,
    .flags = 0,
    .type = EV_KEY,
    .keymap = mahimahi_keypad_key_map,
    .keymap_size = ARRAY_SIZE(mahimahi_keypad_key_map)
};
```

在上述代码中，mahimahi_keypad_key_matrix_info 和 mahimahi_keypad_info 是 gpio_event_matrix_info 类型的结构体，分别实现两个按键和一个按键的处理功能。

(3) 使用 GPIO 驱动实现轨迹球部分驱动，在实现时，由 X 方向和 Y 方向两部分组成。具体代码如下所示：

```
static uint32_t jogball_x_gpios[] = {
   MAHIMAHI_GPIO_BALL_LEFT, MAHIMAHI_GPIO_BALL_RIGHT,
};
static uint32 t jogball y gpios[] = {
   MAHIMAHI_GPIO_BALL_UP, MAHIMAHI_GPIO_BALL_DOWN,
};
static struct jog_axis_info jogball_x_axis = {  // X 轴的内容
   .info = {
      .info.func = gpio_event_axis_func,  // 关键函数实现
      .count = ARRAY_SIZE(jogball_x_gpios),
      .dev = 1,
      .type = EV REL,
      .code = REL_X,
      .decoded_size = 1U << ARRAY_SIZE(jogball_x_gpios),
      .map = jogball_axis_map,
      .gpio = jogball_x_gpios,
      .flags = GPIOEAF_PRINT_UNKNOWN_DIRECTION,
   }
};
static struct jog_axis_info jogball_y_axis = { // Y 轴的内容
   .info = {
      .info.func = gpio_event_axis_func,  // 关键函数实现
      .count = ARRAY_SIZE(jogball_y_gpios)
      .dev = 1,
      .type = EV_REL,
      .code = REL_Y,
      .decoded_size = 1U << ARRAY_SIZE(jogball_y_gpios),
      .map = jogball_axis_map,
      .gpio = jogball_y_gpios,
      .flags = GPIOEAF_PRINT_UNKNOWN_DIRECTION,
   }
};
```

在上述代码中，使用 jog_axis_info 类型的结构体定义了轨迹球，这种设备的类型(type)是相对设备 EV_REL。

注意：除了默认的 AVRCP.kl 和 qwerty.kl 之外，在高通 Mahimahi 平台中新增了文件 h2w_headset.kl 和 mahimahi-keypad.kl。

13.4.3　OMAP高通处理器中的输入驱动实现

1. 触摸屏驱动程序

OMAP 的 Zoom 平台的输入设备包括触摸屏和键盘(Qwerty 全键盘)两种，其中触摸屏驱动程序保存在文件 drivers/input/touchscreen/synaptics_i2c_rmi.c 中，这是一个 I^2C 的触摸屏的驱动程序，与 MSM 的完全相同，在此将不再进行讲解。

2. 键盘驱动程序

在 Zoom 平台中，键盘驱动程序保存在文件 drivers/input/keyboard/twl4030_keypad.c 中，此驱动使用了 I^2C 的接口，驱动本身经过了一次封装处理。文件 twl4030_keypad.c 中的核心内容是中断处理相关的内容，其中函数 do_kp_irq()是标准 Linux 系统的中断的处理函数，其主要代码如下所示：

```
static irqreturn_t do_kp_irq(int irq, void *_kp)
{
    struct twl4030_keypad *kp = _kp;
    u8 reg;
    int ret;
    ret = twl4030_kpread(kp, &reg, KEYP_ISR1, 1); // 调用 twl4030_i2c_read
    if ((ret >= 0) && (reg & KEYP_IMR1_KP))
        twl4030_kp_scan(kp, 0);          // 非释放所有的处理
    else
        twl4030_kp_scan(kp, 1);       // 释放所有的处理
    return IRQ_HANDLED;
}
```

函数 twl4030_kp_scan 负责实现核心处理功能，先负责找到按键的行列，然后调用函数 input_report_key()来汇报结果。函数 twl4030_kp_scan 的主要实现代码如下所示：

```
static void twl4030_kp_scan(struct twl4030_keypad *kp, int release_all)
{
    u16 new_state[MAX_ROWS];
    int col, row;
    // ...... 省略部分内容
    for (row=0; row<kp->n rows; row++) {
        int changed = new_state[row] ^ kp->kp_state[row];
        // ...... 省略部分内容
        for (col=0; col<kp->n_cols; col++) {
            int key;
            key = twl4030_find_key(kp, col, row);
            // ...... 省略部分内容
            input_report_key(kp->input, key,        // 上报按键消息
```

```
                    new_state[row] & (1 << col));
        }
        kp->kp_state[row] = new_state[row];
    }
    input_sync(kp->input);
}
```

接下来，使用函数 twl4030_find_key 根据行列来扫描键盘信息，其实现代码如下所示：

```
static int twl4030_find_key(struct twl4030_keypad *kp, int col, int row)
{
    int i, rc;
    rc = KEY(col, row, 0);
    for (i=0; i<kp->keymapsize; i++)
        if ((kp->keymap[i] & ROWCOL_MASK) == rc)
            return kp->keymap[i] & (KEYNUM_MASK | KEY_PERSISTENT);
    return -EINVAL;
}
```

在此需要注意，在上述代码中使用 kp->keymap 数组定义了按键映射关系，此数组在文件 arch/arm/mach-omap2/board-zoom2.c 中定义，并对应于数组 zoom2_twl4030_keymap，此数组的定义代码如下所示：

```
static int zoom2 twl4030 keymap[] = {
    KEY(0, 0, KEY_E),
    KEY(1, 0, KEY_R),
    KEY(2, 0, KEY_T),
    KEY(3, 0, KEY_HOME),
    KEY(6, 0, KEY_I),
    KEY(7, 0, KEY_LEFTSHIFT),
    ...
    KEY(7, 7, KEY_DOWN),
    KEY(0, 7, KEY_PROG1),
    KEY(1, 7, KEY_PROG2),
    KEY(2, 7, KEY_PROG3),
    KEY(3, 7, KEY_PROG4),
    0
};
```

在 OMAP 的 Zoom 平台中，因为键盘基本上是全键盘，并且其数字键和字母键是共用的，所以使用全键盘的配置文件基本上可以实现全部功能。

Android

第 14 章 蓝牙系统详解

蓝牙是一种支持设备短距离(一般 10m 内)通信的无线电技术，可以在包括移动电话、PDA、无线耳机、笔记本电脑、相关外设等众多设备之间进行无线信息交换。

在本章的内容中，将讲解 Android 4.3 系统中蓝牙模块的底层源码和实现原理，为读者步入本书后面知识的学习打下基础。

14.1 Android系统中的蓝牙模块

Android 系统包含了对蓝牙网络协议栈的支持，这使得蓝牙设备能够无线连接其他蓝牙设备并交换数据。Android 的应用程序框架提供了访问蓝牙功能的 APIs。这些 APIs 让应用程序能够无线连接其他蓝牙设备，实现点对点，或点对多点的无线交互功能。

通过使用蓝牙 APIs，一个 Android 应用程序能够实现如下所示的功能：

- 扫描其他蓝牙设备。
- 查询本地蓝牙适配器(Local Bluetooth Adapter)，用于配对蓝牙设备。
- 建立 RFCOMM 信道(channels)。
- 通过服务发现(Service Discovery)连接其他设备。
- 数据通信。
- 管理多个连接。

Android 平台的蓝牙系统是基于 BlueZ 实现的，是通过 Linux 中一套完整的蓝牙协议栈开源实现的。当前 BlueZ 被广泛应用于各种 Linux 版本中，并被芯片公司移植到各种芯片平台上使用。在 Linux 2.6 内核中，已经包含了完整的 BlueZ 协议栈，在 Android 系统中已经移植并嵌入进了 BlueZ 的用户空间实现，并且随着硬件技术的发展而不断更新。

蓝牙(Bluetooth)技术实际上是一种短距离无线电技术。在 Android 系统的蓝牙模块中，除了使用 Kernel 支持外，还需要用户空间的 BlueZ 的支持。

Android 平台中蓝牙模块的基本层次结构如图 14-1 所示。

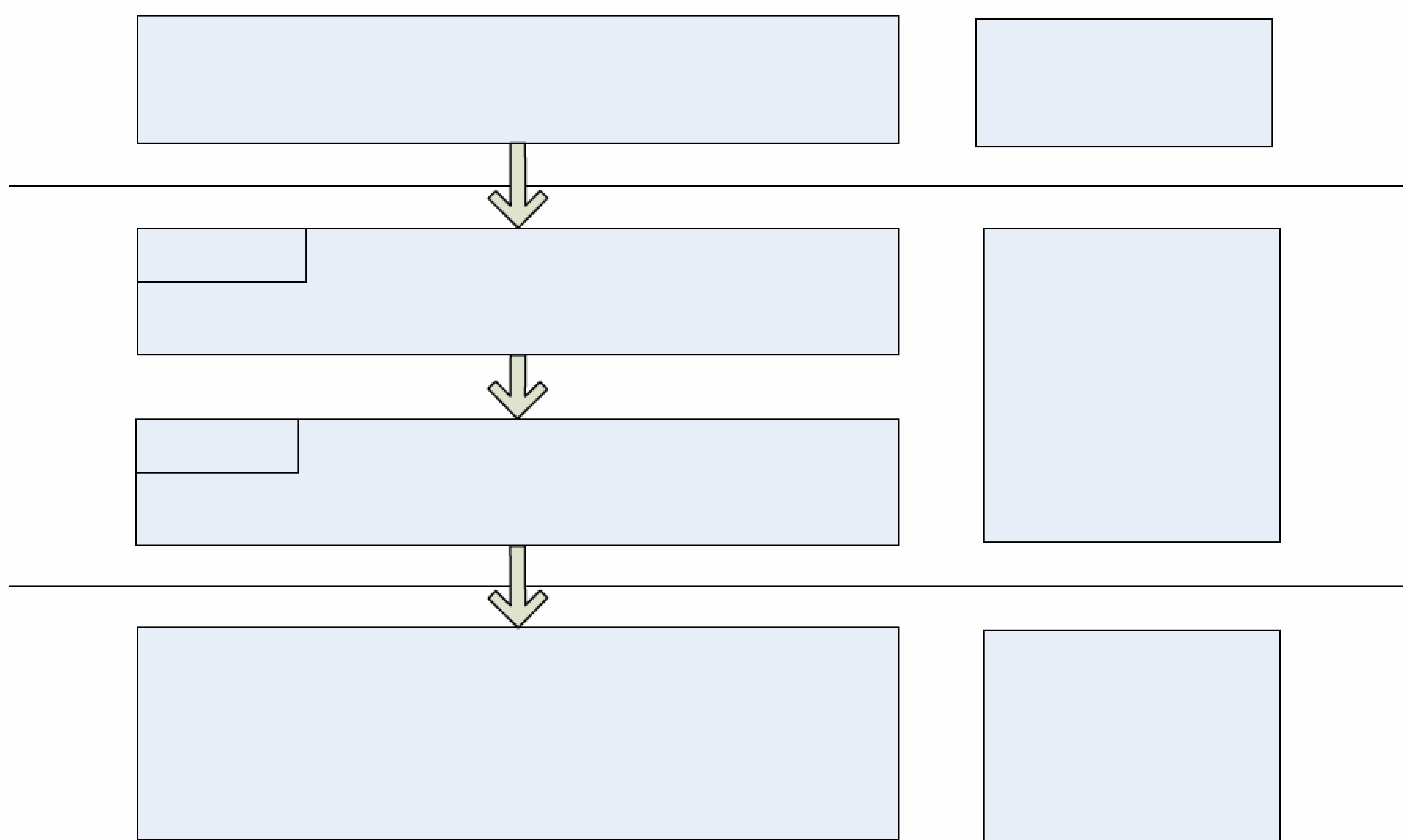


图 14-1　蓝牙系统的层次结构

在 Android 平台中，蓝牙系统从上到下主要包括 Java 框架中的 BlueTooth 类、Android 适配库、BlueZ 库、驱动程序和协议，这几部分的系统结构如图 14-2 所示。

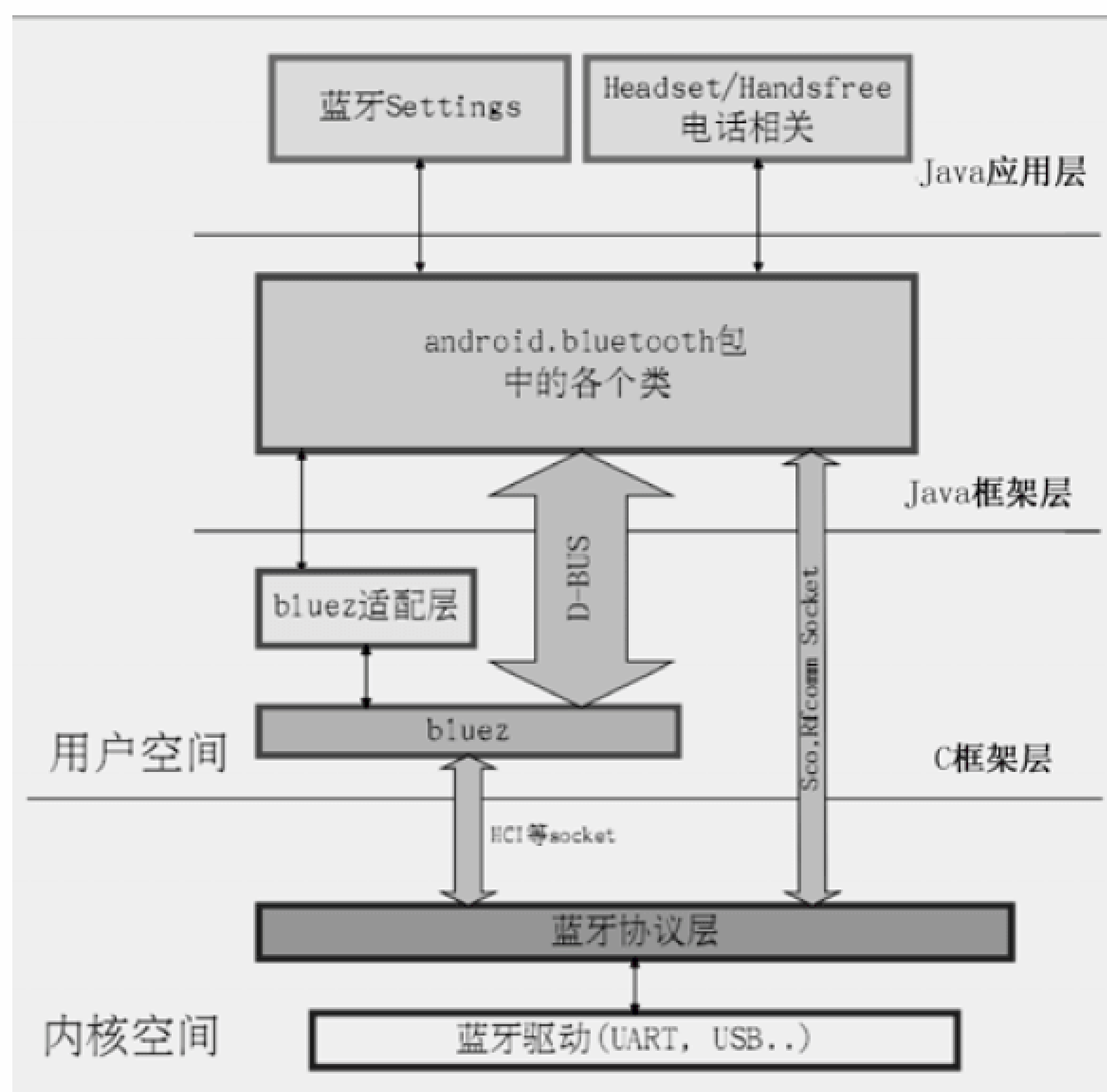


图 14-2　蓝牙系统的结构

在图 14-2 中，各个层次结构的具体说明如下所示。

(1)　BlueZ 库

Android 蓝牙设备管理的库的路径如下所示：

```
external/bluez/
```

可以分别生成库 libbluetooth.so、libbluedroid.so 和 hcidump 等众多的相关工具和库。BlueZ 库提供了对用户空间蓝牙的支持，在里面包含了主机控制协议 HCI 以及其他众多内核实现协议的接口，并且实现了所有蓝牙应用模式 Profile。

(2)　蓝牙的 JNI 部分

此部分的代码路径如下所示：

```
frameworks/base/core/jni/
```

(3)　Java 框架层

Java 框架层的实现代码保存在如下路径中：

```
frameworks/base/core/java/android/bluetooth         //蓝牙部分对应应用程序的 API
frameworks/base/core/java/android/Server            //蓝牙的服务部分
```

蓝牙的服务部分负责管理并使用底层本地服务，并封装成系统服务。而在 android.bluetooth 部分中，包含了各个蓝牙平台的 API 部分，以供应用程序层使用。

(4)　BlueTooth 的适配库

BlueTooth 适配库的代码路径如下所示：

```
system/bluetooth/
```

此层用于生成库 libbluedroid.so 以及相关的工具和库，能够实现对蓝牙设备的管理，例如蓝牙设备的电源管理。

14.2 分析蓝牙模块的源码

要想掌握蓝牙系统的开发原理，需要首先分析 Android 中的蓝牙源码并了解其核心构造，只有这样，才能对蓝牙应用开发做到游刃有余。在本节的内容中，将简要介绍开源 Android 中蓝牙模块相关的代码，为读者步入本书后面知识的学习打下基础。

14.2.1 初始化蓝牙芯片

初始化蓝牙芯片的工作是通过 BlueZ 的 hciattach 工具进行的，此工具在如下目录的文件中实现：

```
external/bluetooth/tools
```

hciattach 命令主要用来初始化蓝牙设备，它的命令格式如下所示：

```
hciattach [-n] [-p] [-b] [-t timeout] [-s initial_speed] <tty> <type | id> [speed]
[flow|noflow][bdaddr]
```

在上述格式中，最重要的参数就是 type 和 speed，type 决定了要初始化的设备的型号，可以使用 hciattach -l 来列出所支持的设备型号。

并不是所有的参数对所有的设备都是适用的，有些设备会忽略一些参数设置，例如，查看 hciattach 的代码就可以看到，多数设备都忽略 bdaddr 参数。hciattach 命令内部的工作步骤是：首先打开指定的 tty 设备，然后做一些通用的设置，如 flow 等，然后设置波特率为 initial_speed，然后根据 type 调用各自的初始化代码，最后将波特率重新设置为 speed。所以调用 hciattach 时，要根据你的实际情况，设置好 initial_speed 和 speed。

对于 type BCSP 来说，它的初始化代码只做一件事，就是完成 BCSP 协议的同步操作，它并不对蓝牙芯片做任何的 pskey 设置。

14.2.2 蓝牙服务

在蓝牙服务方面一般不要我们自己定义，只需要使用初始化脚本文件 init.rc 中的默认内容即可。例如下面的代码：

```
service bluetoothd /system/bin/logwrapper /system/bin/bluetoothd -d -n
   socket bluetooth stream 660 bluetooth bluetooth
   socket dbus_bluetooth stream 660 bluetooth bluetooth
   # init.rc does not yet support applying capabilities, so run as root and
   # let bluetoothd drop uid to bluetooth with the right linux capabilities
   group bluetooth net_bt_admin misc
   disabled

# baudrate change 115200 to 1152000(Bluetooth)
service changebaudrate /system/bin/logwrapper /system/xbin/bccmd_115200 -t bcsp
-d /dev/s3c2410_serial1 psset -r 0x1be 0x126e
```

```
    user bluetooth
    group bluetooth net bt admin
    disabled
    oneshot

#service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 1152000
/dev/s3c2410_serial1 bcsp 1152000
service hciattach /system/bin/logwrapper /system/bin/hciattach -n -s 115200
/dev/s3c2410_serial1 bcsp 115200
    user bluetooth
    group bluetooth net_bt_admin misc
    disabled

service hfag /system/bin/sdptool add --channel=10 HFAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service hsag /system/bin/sdptool add --channel=11 HSAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service opush /system/bin/sdptool add --channel=12 OPUSH
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service pbap /system/bin/sdptool add --channel=19 PBAP
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot
```

在上述代码中，每一个 service 后面都列出了一种 Android 服务。

14.2.3 管理蓝牙电源

在 Android 系统的如下目录中实现了 libbluedroid：

```
system/bluetooth/
```

我们可以调用 rfkill 接口来控制电源管理，如果已经实现了 rfkill 接口，则无须再进行配置。如果在文件 init.rc 中已经实现了 hciattach 服务，则说明在 libbluedroid 中已经实现了对其调用以操作蓝牙的初始化。

14.3　与蓝牙相关的类

经过本章前面内容的学习，我们已经了解了 Android 系统中蓝牙的基本知识，了解了蓝牙的工作原理和机制。在本节的内容中，将详细讲解 Android 系统中与蓝牙相关的类，为读者步入本书后面知识的学习打好基础。

14.3.1　BluetoothSocket类

1. BluetoothSocket类的基础

Android 的蓝牙系统和 Socket 套接字密切相关，蓝牙端的监听接口与 TCP 的端口类似，都是使用了 Socket 和 ServerSocket 类。在服务器端，使用 BluetoothServerSocket 类来创建一个监听服务端口。若一个连接被 BluetoothServerSocket 所接受，它会返回一个新的 BluetoothSocket 来管理该连接。在客户端，使用一个单独的 BluetoothSocket 类去初始化一个外接连接和管理该连接。

最通常使用的蓝牙端口是 RFCOMM，它是被 Android API 支持的类型。RFCOMM 是一个面向连接，通过蓝牙模块进行数据流传输的方式。为了创建一个 BluetoothSocket 去连接到一个已知设备，使用方法 BluetoothDevice.createRfcommSocketToServiceRecord()。然后调用 connect()方法去尝试一个面向远程设备的连接。这个调用将被阻塞，直到一个连接已经建立，或者该链接失效。

为了创建一个 BluetoothSocket 作为服务端(或者“主机”)，每当该端口连接成功后，无论它初始化为客户端，或者被接受作为服务器端，都通过方法 getInputStream()和 getOutputStream()来打开 IO 流，从而获得各自的 InputStream 和 OutputStream 对象

BluetoothSocket 类的线程是安全的，因为 close()方法总会马上放弃外界操作并关闭服务器端口。

2. BluetoothSocket类的公共方法

(1)　public void close()

功能：马上关闭该端口并且释放所有相关的资源。在其他线程的该端口中引起阻塞，从而使系统马上抛出一个 IO 异常。

异常：IOException。

(2)　public void connect()

功能：尝试连接到远程设备。该方法将阻塞，直到一个连接建立或者失效。如果该方法没有返回异常值，则该端口现在已经建立。当设备查找正在进行的时候，创建对远程蓝牙设备的新连接不可被尝试。设备查找在蓝牙适配器上是一个重量级过程，并且肯定会降低一个设备的连接。使用 cancelDiscovery()方法会取消一个外界的查询，因为这个查询并不由活动所管理，而是作为一个系统服务来运行，所以即使它不能直接请求一个查询，应用程序也总会调用 cancelDiscovery()方法。方法 close()可以用来放弃从另一线程而来的调用。

异常：IOException，表示一个错误，例如连接失败。

(3)　public InputStream getInputStream()

功能：通过连接的端口获得输入数据流。即使该端口未连接，该输入数据流也会返回，不过在该数据流上的操作将抛出异常，直到相关的连接已经建立。

返回值：输入流。

异常：IOException。

(4)　public OutputStream getOutputStream()

功能：通过连接的端口获得输出数据流。即使该端口未连接，该输出数据流也会返回，不过在该数据流上的操作将抛出异常，直到相关的连接已经建立。

返回值：输出流。

异常：IOException。

(5)　public BluetoothDevice getRemoteDevice()

功能：获得该端口正在连接或者已经连接的远程设备。

返回值：远程设备。

14.3.2　BluetoothServerSocket类

1. BluetoothServerSocket类基础

类 BluetoothServerSocket 的格式如下所示：

```
public final class BluetoothServerSocket extends Object implements Closeable
```

类 BluetoothServerSocket 的结构如下所示：

```
java.lang.Object
  android.bluetooth.BluetoothServerSocket
```

2. BluetoothServerSocket类的公共方法

(1)　public BluetoothSocket accept(int timeout)

功能：阻塞直到超时时间内的连接建立。在一个成功建立的连接上返回一个已连接的 BluetoothSocket 类。每当该调用返回时，它可以再次调用去接收以后新来的连接。close()方法可以用来放弃从另一线程来的调用。

参数 timeout：表示阻塞超时时间。

返回值：已连接的 BluetoothSocket。

异常：IOException，表示出现错误，比如该调用被放弃或超时。

(2)　public BluetoothSocket accept()

功能：阻塞直到一个连接已经建立。在一个成功建立的连接上返回一个已连接的 BluetoothSocket 类。每当该调用返回的时候，它可以再次调用去接收以后新来的连接。使用 close()方法可以放弃从另一线程来的调用。

返回值：已连接的 BluetoothSocket。

异常：IOException，表示出现错误，比如该调用被放弃或者超时。

(3) public void close()

功能：马上关闭端口，并释放所有相关的资源。在其他线程的该端口中引起阻塞，从而使系统马上抛出一个 IO 异常。

关闭 BluetoothServerSocket 不会关闭接受自 accept()的任意 BluetoothSocket。

异常：IOException。

14.3.3 BluetoothAdapter类

1. BluetoothAdapter类基础

类 BluetoothAdapter 的格式如下所示：

```
public final class BluetoothAdapter extends Object
```

类 BluetoothAdapter 的结构如下所示：

```
java.lang.Object
  android.bluetooth.BluetoothAdapter
```

BluetoothAdapter 代表本地的蓝牙适配器设备，通过此类，可以让用户能执行基本的蓝牙任务。例如初始化设备的搜索，查询可匹配的设备集，使用一个已知的 MAC 地址来初始化一个 BluetoothDevice 类，创建一个 BluetoothServerSocket 类以监听其他设备对本机的连接请求等。

为了得到这个代表本地蓝牙适配器的 BluetoothAdapter 类，需要调用静态方法 getDefaultAdapter()，这是所有蓝牙动作使用的第一步。当拥有本地适配器以后，用户可以获得一系列的 BluetoothDevice 对象，这些对象代表所有拥有 getBondedDevice()方法的已经匹配的设备；用 startDiscovery()方法来开始设备的搜寻；或者创建一个 BluetoothServerSocket 类，通过 listenUsingRfcommWithServiceRecord(String, UUID)方法来监听新来的连接请求。

注意：大部分方法需要 BLUETOOTH 权限，一些方法同时需要 BLUETOOTH_ADMIN 权限。

2. BluetoothAdapter类的常量

(1) String ACTION_DISCOVERY_FINISHED

广播事件：本地蓝牙适配器已经完成设备的搜寻过程。需要 BLUETOOTH 权限接收。

常量值：android.bluetooth.adapter.action.DISCOVERY_FINISHED。

(2) String ACTION_DISCOVERY_STARTED

广播事件：本地蓝牙适配器已经开始了对远程设备的搜寻过程。它通常牵涉到一个大概需时 12 秒的查询扫描过程，紧跟着是一个对每个获取到自身蓝牙名称的新设备的页面扫描。用户会发现一个把 ACTION_FOUND 常量通知为远程蓝牙设备的注册。设备查找是一个重量级过程。当查找正在进行的时候，用户不能尝试对新的远程蓝牙设备进行连接，同时存在的连接将获得有限制的带宽以及高等待时间。用户可用 cancelDiscovery()类来取消正在执行的查找进程。需要 BLUETOOTH 权限接收。

常量值：android.bluetooth.adapter.action.DISCOVERY_STARTED。

(3)　String ACTION_LOCAL_NAME_CHANGED

广播活动：本地蓝牙适配器已经更改了它的蓝牙名称。该名称对远程蓝牙设备是可见的，它总是包含一个带有名称的 EXTRA_LOCAL_NAME 附加域。需要 BLUETOOTH 权限接收。

常量值：android.bluetooth.adapter.action.LOCAL_NAME_CHANGED。

(4)　String ACTION_REQUEST_DISCOVERABLE

Activity 活动：显示一个请求被搜寻模式的系统活动。如果蓝牙模块当前未打开，该活动也将请求用户打开蓝牙模块。被搜寻模式与 SCAN_MODE_CONNECTABLE_DISCOVERABLE 等价。当远程设备执行查找进程的时候，它允许其发现该蓝牙适配器。从隐私安全考虑，Android 不会将被搜寻模式设置为默认状态。

该意图的发送者可以选择性地运用 EXTRA_DISCOVERABLE_DURATION 这个附加域去请求发现设备的持续时间。普遍来说，对于每一请求，默认的持续时间为 120 秒，最大值则可达到 300 秒。

Android 运用 onActivityResult(int, int, Intent)回收方法来传递该活动结果的通知。被搜寻的时间(以秒为单位)将通过 resultCode 值来显示，如果用户拒绝被搜寻，或者设备产生了错误，则通过 RESULT_CANCELED 值来显示。

每当扫描模式变化的时候，应用程序可以通过 ACTION_SCAN_MODE_CHANGED 值来监听全局的消息通知。比如，当设备停止被搜寻以后，该消息可以被系统通知給应用程序。需要 BLUETOOTH 权限。

常量值：android.bluetooth.adapter.action.REQUEST_DISCOVERABLE。

(5)　String ACTION_REQUEST_ENABLE

Activity 活动：显示一个允许用户打开蓝牙模块的系统活动。当蓝牙模块完成打开工作，或者当用户决定不打开蓝牙模块时，系统活动将返回该值。Android 运用 onActivityResult(int, int, Intent)回收方法来传递该活动结果的通知。如果蓝牙模块被打开，将通过 resultCode 值 RESULT_OK 来显示；如果用户拒绝该请求或设备产生了错误，则通过 RESULT_CANCELED 值来显示。每当蓝牙模块被打开或者关闭，应用程序可以通过 ACTION_STATE_CHANGED 值来监听全局的消息通知。需要 BLUETOOTH 权限

常量值：android.bluetooth.adapter.action.REQUEST_ENABLE。

(6)　String ACTION_SCAN_MODE_CHANGED

广播活动：指明蓝牙扫描模块或者本地适配器已经发生变化。它总是包含 EXTRA_SCAN_MODE 和 EXTRA_PREVIOUS_SCAN_MODE。这两个附加域各自包含了新的和旧的扫描模式。需要 BLUETOOTH 权限。

常量值：android.bluetooth.adapter.action.SCAN_MODE_CHANGED。

(7)　String ACTION_STATE_CHANGED

广播活动：本来的蓝牙适配器的状态已经改变，例如蓝牙模块已经被打开或者关闭。它总是包含 EXTRA_STATE 和 EXTRA_PREVIOUS_STATE。这两个附加域各自包含了新的和旧的状态。需要 BLUETOOTH 权限接收。

常量值：android.bluetooth.adapter.action.STATE_CHANGED。

(8)　int ERROR

功能：标记该类的错误值。确保与该类中的任意其他整数常量不相等。它为需要一个标记

错误值的函数提供了便利。例如：

```
Intent.getIntExtra(BluetoothAdapter.EXTRA_STATE, BluetoothAdapter.ERROR)
```

常量值：-2147483648 (0x80000000)。

(9) String EXTRA_DISCOVERABLE_DURATION

功能：试图在 ACTION_REQUEST_DISCOVERABLE 常量中作为一个可选的整型附加域，来为短时间内的设备发现请求一个特定的持续时间。默认值为 120 秒，超过 300 秒的请求将被限制。这些值是可以变化的。

常量值：android.bluetooth.adapter.extra.DISCOVERABLE_DURATION。

(10) String EXTRA_LOCAL_NAME

功能：试图在 ACTION_LOCAL_NAME_CHANGED 常量中作为一个字符串附加域，来请求本地蓝牙的名称。

常量值：android.bluetooth.adapter.extra.LOCAL_NAME。

(11) String EXTRA_PREVIOUS_SCAN_MODE

功能：试图在 ACTION_SCAN_MODE_CHANGED 常量中作为一个整型附加域，来请求以前的扫描模式。可能值有如下几种：

- SCAN_MODE_NONE
- SCAN_MODE_CONNECTABLE
- SCAN_MODE_CONNECTABLE_DISCOVERABLE

常量值：android.bluetooth.adapter.extra.PREVIOUS_SCAN_MODE。

(12) String EXTRA_PREVIOUS_STATE

功能：试图在 ACTION_STATE_CHANGED 常量中作为一个整型附加域，来请求以前的供电状态。可以取的值如下所示：

- STATE_OFF
- STATE_TURNING_ON
- STATE_ON
- STATE_TURNING_OFF

常量值：android.bluetooth.adapter.extra.PREVIOUS_STATE。

(13) String EXTRA_SCAN_MODE

功能：试图在 ACTION_SCAN_MODE_CHANGED 常量中作为一个整型附加域，来请求当前的扫描模式。可以取的值如下所示：

- SCAN_MODE_NONE
- SCAN_MODE_CONNECTABLE
- SCAN_MODE_CONNECTABLE_DISCOVERABLE

常量值：android.bluetooth.adapter.extra.SCAN_MODE。

(14) String EXTRA_STATE

功能：试图在 ACTION_STATE_CHANGED 常量中作为一个整型附加域，来请求当前的供电状态。可以取的值如下所示：

- STATE_OFF
- STATE_TURNING_ON

- STATE_ON
- STATE_TURNING_OFF

常量值：android.bluetooth.adapter.extra.STATE。

(15) int SCAN_MODE_CONNECTABLE

功能：指明在本地蓝牙适配器中，查询扫描功能失效，但页面扫描功能有效。因此该设备不能被远程蓝牙设备发现，但如果以前曾经发现过该设备，则远程设备可以对其进行连接。

常量值：21 (0x00000015)。

(16) int SCAN_MODE_CONNECTABLE_DISCOVERABLE

功能：指明在本地蓝牙适配器中，查询扫描功能和页面扫描功能都有效。因此该设备既可以被远程蓝牙设备发现，也可以被其连接。

常量值：23 (0x00000017)。

(17) int SCAN_MODE_NONE

功能：指明在本地蓝牙适配器中，查询扫描功能和页面扫描功能都失效. 因此该设备既不可以被远程蓝牙设备发现，也不可以被其连接。

常量值：20 (0x00000014)。

(18) int STATE_OFF

功能：指明本地蓝牙适配器模块已经关闭。

常量值：10 (0x0000000a)。

(19) int STATE_ON

功能：指明本地蓝牙适配器模块已经打开，并且准备被使用。

(20) int STATE_TURNING_OFF

功能：指明本地蓝牙适配器模块正在关闭。本地客户端可以立刻尝试友好地断开任意外部连接。

常量值：13 (0x0000000d)。

(21) int STATE_TURNING_ON

功能：指明本地蓝牙适配器模块正在打开。然而本地客户在尝试使用这个适配器之前需要为 STATE_ON 状态而等待。

常量值：11 (0x0000000b)。

3. BluetoothAdapter类的公共方法

(1)　public boolean cancelDiscovery()

功能：取消当前的设备发现查找进程，需要 BLUETOOTH_ADMIN 权限。因为对蓝牙适配器而言，查找是一个重量级的过程，因此该方法必须在尝试连接到远程设备前使用 connect() 方法进行调用。发现的过程不会由活动来进行管理，但是它会作为一个系统服务来运行，因此即使它不能直接请求这样的一个查询动作，也必需取消该搜索进程。如果蓝牙状态不是 STATE_ON，这个 API 将返回 false。蓝牙打开后，等待 ACTION_STATE_CHANGED 更新成 STATE_ON。

返回值：若成功则返回 true，有错误则返回 false。

(2) public static boolean checkBluetoothAddress(String address)

功能：验证诸如“00:43:A8:23:10:F0”之类的蓝牙地址，字母必须为大写才有效。

参数 address：字符串形式的蓝牙模块地址。

返回值：若地址正确则返回 true，否则返回 false。

(3) public boolean disable()

功能：关闭本地蓝牙适配器——不能在没有明确关闭蓝牙的用户动作中使用。这个方法友好地停止所有的蓝牙连接，停止蓝牙系统服务，以及对所有基础蓝牙硬件进行断电。没有用户的直接同意，蓝牙永远不能被禁止。这个 disable()方法只提供了一个应用，该应用包含了一个改变系统设置的用户界面(例如“电源控制”应用)。

这是一个异步调用方法：该方法将马上获得返回值，用户要通过监听 ACTION_STATE_CHANGED 值来获取随后的适配器状态改变的通知。如果该调用返回 true 值，则该适配器状态会立刻从 STATE_ON 转向 STATE_TURNING_OFF，稍后则会转为 STATE_OFF 或者 STATE_ON。如果该调用返回 false，那么系统已经有一个保护蓝牙适配器被关闭的问题——比如该适配器已经被关闭了。

需要 BLUETOOTH_ADMIN 权限。

返回值：如果蓝牙适配器的停止进程已经开启，则返回 true，如果产生错误，则返回 false。

(4) public boolean enable()

功能：打开本地蓝牙适配器——不能在没有明确打开蓝牙的用户动作中使用。该方法将为基础的蓝牙硬件供电，并且启动所有的蓝牙系统服务。没有用户的直接同意，蓝牙永远不能被禁止。如果用户为了创建无线连接而打开了蓝牙模块，则需要 ACTION_REQUEST_ENABLE 值，该值将提出一个请求用户允许以打开蓝牙模块的会话。这个 enable()值只提供了一个应用，该应用包含了一个改变系统设置的用户界面(例如“电源控制”应用)。

这是一个异步调用方法：该方法将马上获得返回值，用户要通过监听 ACTION_STATE_CHANGED 值来获取随后的适配器状态改变的通知。如果该调用返回 true 值，则该适配器状态会立刻从 STATE_OFF 转向 STATE_TURNING_ON，稍后则会转为 STATE_OFF 或者 STATE_ON。

如果该调用返回 false，那么说明系统已经有一个保护蓝牙适配器被打开的问题——比如飞行模式，或者该适配器已经被打开。

该方法需要 BLUETOOTH_ADMIN 权限。

返回值：如果蓝牙适配器的打开进程已经开启，则返回 true；如果产生错误，则返回 false。

(5) public String getAddress()

功能：返回本地蓝牙适配器的硬件地址，例如：

```
00:11:22:AA:BB:CC
```

该方法需要 BLUETOOTH 权限。

返回值：字符串形式的蓝牙模块地址。

(6) public Set<BluetoothDevice> getBondedDevices()

功能：返回已经匹配到本地适配器的 BluetoothDevice 类的对象集合。如果蓝牙状态不是 STATE_ON，这个 API 将返回 false。蓝牙打开后，等待 ACTION_STATE_CHANGED 更新成 STATE_ON。需要 BLUETOOTH 权限。

返回值：未被修改的 BluetoothDevice 类的对象集合，如果有错误，则返回 null。

(7) public static synchronized BluetoothAdapter getDefaultAdapter()

功能：获取对默认本地蓝牙适配器的操作权限。目前 Andoird 只支持一个蓝牙适配器，但是 API 可以被扩展为支持多个适配器。该方法总是返回默认的适配器。

返回值:返回默认的本地适配器,如果蓝牙适配器在该硬件平台上不能被支持,则返回 null。

(8) public String getName()

功能：获取本地蓝牙适配器的蓝牙名称，这个名称对于外界蓝牙设备而言是可见的。需要 BLUETOOTH 权限。

返回值：该蓝牙适配器名称，如果有错误，则返回 null。

(9) public BluetoothDevice getRemoteDevice(String address)

功能：为给予的蓝牙硬件地址获取一个 BluetoothDevice 对象。合法的蓝牙硬件地址必须为大写，格式类似于“00:11:22:33:AA:BB”。checkBluetoothAddress(String)方法可以用来验证蓝牙地址的正确性。BluetoothDevice 类对于合法的硬件地址总会产生返回值，即使这个适配器从未见过该设备。

参数：address 为合法的蓝牙 MAC 地址。

异常：IllegalArgumentException，如果地址不合法就抛出。

(10) public int getScanMode()

功能：获取本地蓝牙适配器的当前蓝牙扫描模式，蓝牙扫描模式决定本地适配器可连接并且/或者可被远程蓝牙设备所连接。需要 BLUETOOTH 权限，可能的取值如下所示：

- SCAN_MODE_NONE
- SCAN_MODE_CONNECTABLE
- SCAN_MODE_CONNECTABLE_DISCOVERABLE

如果蓝牙状态不是 STATE_ON，则这个 API 将返回 false。蓝牙打开后，等待 ACTION_STATE_CHANGED 更新成 STATE_ON。

返回值：扫描模式。

(11) public int getState()

功能：获取本地蓝牙适配器的当前状态，需要 BLUETOOTH 类。可能的取值如下所示：

- STATE_OFF
- STATE_TURNING_ON
- STATE_ON
- STATE_TURNING_OFF

返回值：蓝牙适配器的当前状态。

(12) public boolean isDiscovering()

功能：如果当前蓝牙适配器正处于设备发现查找进程中，则返回真值。设备查找是一个重量级过程。当查找正在进行的时候，用户不能尝试对新的远程蓝牙设备进行连接，同时存在的连接将获得有限制的带宽以及较长的等待时间。用户可用 cencelDiscovery()类来取消正在执行的查找进程。

应用程序也可以为 ACTION_DISCOVERY_STARTED 或者 ACTION_DISCOVERY_FINISHED 进行注册，从而当查找开始或者完成的时候，可以获得通知。

如果蓝牙状态不是STATE_ON，这个API将返回false。蓝牙打开后，等待ACTION_STATE_CHANGED 更新成 STATE_ON。需要 BLUETOOTH 权限。

返回值：如果正在查找，则返回 true。

(13) public boolean isEnabled()

功能：如果蓝牙正处于打开状态并可用，则返回真值，与 getBluetoothState()==STATE_ON 等价，需要 BLUETOOTH 权限。

返回值：如果本地适配器已经打开，则返回 true。

(14) public BluetoothServerSocket listenUsingRfcommWithServiceRecord(String name, UUID uuid)

功能：创建一个正在监听的安全的带有服务记录的无线射频通信(RFCOMM)蓝牙端口。一个对该端口进行连接的远程设备将被认证，对该端口的通信将被加密。使用 accept()方法可以获取从监听 BluetoothServerSocket 处新来的连接。该系统分配一个未被使用的无线射频通信通道来进行监听。

该系统也将注册一个服务探索协议(SDP)记录，该记录带有一个包含了特定的通用唯一识别码(Universally Unique Identifier，UUID)、服务器名称和自动分配通道的本地 SDP 服务。远程蓝牙设备可以用相同的 UUID 来查询自己的 SDP 服务器，并搜寻连接到了哪个通道上。如果该端口已经关闭，或者如果该应用程序异常退出，则这个 SDP 记录会被移除。使用 createRfcommSocketToServiceRecord(UUID)可以从另一个使用相同 UUID 的设备来连接到这个端口。需要 BLUETOOTH 权限。

参数如下。

- name：SDP 记录下的服务器名。
- uuid：SDP 记录下的 UUID。

返回值：一个正在监听的无线射频通信蓝牙服务端口。

异常：IOException，表示产生错误，如蓝牙设备不可用，或许可无效，或通道被占用。

(15) public boolean setName(String name)

功能：设置蓝牙或者本地蓝牙适配器的昵称，这个名字对于外界蓝牙设备而言是可见的。合法的蓝牙名称最多拥有 248 位 UTF-8 字符，但是很多外界设备只能显示前 40 个字符，有些可能只限制前 20 个字符。

如果蓝牙状态不是STATE_ON，这个API将返回false。蓝牙打开后，等待ACTION_STATE_CHANGED 更新成 STATE_ON。需要 BLUETOOTH_ADMIN 权限。

参数 name：一个合法的蓝牙名称。

返回值：如果该名称已被设定，则返回 true，否则返回 false。

(16) public boolean startDiscovery()

功能：开始对远程设备进行查找的进程，它通常牵涉到一个大概需时 12 秒的查询扫描过程，紧跟着是一个对每个获取到自身蓝牙名称的新设备的页面扫描。

这是一个异步调用方法：该方法将马上获得返回值，注册 ACTION_DISCOVERY_STARTED 和 ACTION_DISCOVERY_FINISHED，意图准确地确定该探索是处于开始阶段或者完成阶段。注册 ACTION_FOUND 以获得远程蓝牙设备已找到的通知。

发现的过程不会由活动来进行管理，但是它会作为一个系统服务来运行，因此即使它不能

直接请求这样的一个查询动作，也必需取消该搜索进程。设备搜寻只寻找已经被连接的远程设备。许多蓝牙设备默认不会被搜寻到，并且需要进入到一个特殊的模式中。

如果蓝牙状态不是 STATE_ON，这个 API 将返回 false。蓝牙打开后，等待 ACTION_STATE_CHANGED 更新成 STATE_ON。需要 BLUETOOTH_ADMIN 权限。

返回值：成功返回 true，错误返回 false。

14.3.4 BluetoothClass.Service类

类 BluetoothClass.Service 的格式如下所示：

```
public static final class BluetoothClass.Service extends Object
```

类 BluetoothClass.Service 的结构如下所示：

```
java.lang.Object
  android.bluetooth.BluetoothClass.Service
```

类 BluetoothClass.Service 用于定义所有的服务类常量，任意 BluetoothClass 由 0 或多个服务类编码组成。在类 BluetoothClass.Service 中包含如下所示的常量：

- int AUDIO
- int CAPTURE
- int INFORMATION
- int LIMITED_DISCOVERABILITY
- int NETWORKING
- int OBJECT_TRANSFER
- int POSITIONING
- int RENDER
- int TELEPHONY

14.3.5 BluetoothClass.Device类

类 BluetoothClass.Device 的格式如下所示：

```
public final class BluetoothClass.Device extends Object
```

类 BluetoothClass.Device 的结构如下所示：

```
java.lang.Object
  android.bluetooth.BluetoothClass.Device
```

类 BluetoothClass.Device 用于定义所有的设备类的常量，每个 BluetoothClass 有一个带有主要和较小部分的设备类进行编码。里面的常量代表主要和较小的设备类部分(完整的设备类)的组合。BluetoothClass.Device.Major 的常量只能代表主要设备类，各个常量如下所示。

BluetoothClass.Device 有一个内部类，此内部类定义了所有的主要设备类常量。内部类的定义格式如下所示：

```
class BluetoothClass.Device.Major
```

注意：到此为止，Android 中的蓝牙类介绍完毕。我们在调用这些类时，首先要确保 API Level 至少为版本 5 以上，并且还需注意添加相应的权限，比如在使用通信功能时，需要在文件 androidmanifest.xml 中加入<uses-permission android:name= “android.permission.BLUETOOTH”/>权限，而在开关蓝牙时，需要加入 android. permission.BLUETOOTH_ADMIN 权限。

14.4 低功耗蓝牙协议栈详解

从 Android 4.2 版本开始，Google 便更换了 Android 的蓝牙协议栈，从 BlueZ 换成 BlueDroid。从 Android 4.3 版本开始，提供了对蓝牙 4.0 BLE 的支持。在本节的内容中，将详细讲解 Android 系统中的蓝牙 4.0 BLE 的基本知识，为读者步入本书后面知识的学习打下基础。

14.4.1 低功耗蓝牙协议栈基础

为了确保 Android 系统可以更好地支持蓝牙 4.0 BLE，Broadcom 公司特意推出了适应于 Android 平台的开源低功耗蓝牙协议栈 BlueDroid，其开发文档和 API 是开源代码，在如下所示的地址保存：

```
https://github.com/briandbl/framework
```

在上述开源代码中，低功耗蓝牙 API 支持 Android 平台上的低功耗蓝牙通信功能。通过使用 BlueDroid 协议栈，Android 应用程序可以枚举、发现并访问低功耗蓝牙的外部设备，并且实现了低功耗蓝牙规范。

从 Android 4.2 版本开始，低功耗蓝牙模块的整体结构如图 14-3 所示。

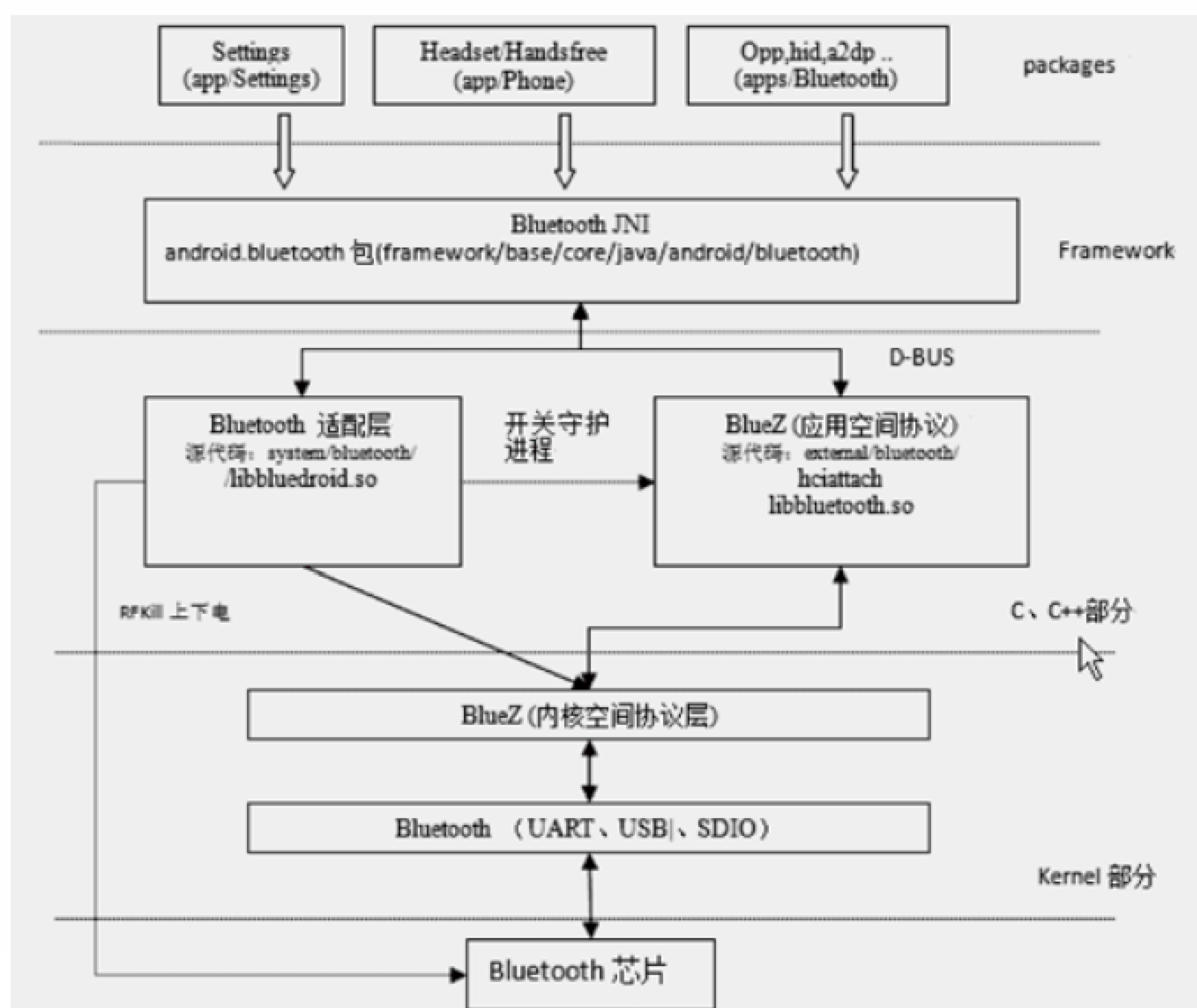


图 14-3 低功耗蓝牙模块的整体结构

注意：虽然从 Android 4.2 版本开始，JNI 部分的代码在 packages 层中实现。但是为了便于读者从视觉上更加容易接受，所以将 JNI 部分绘制在了 Framework 层中。

14.4.2　低功耗蓝牙API详解

Broadcom 公司推出的低功耗蓝牙协议栈 BlueDroid 的开发文档和 API 是开源代码，被保存在如下所示的地址：

```
https://github.com/briandbl/framework
```

在接下来的内容中，将详细讲解主要 API 的基本功能和具体原理。

(1)　本地蓝牙适配器设备

本功能不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，源码位于如下所示的目录中：

```
framework/base/core/java/android.bluetooth/BluetoothAdapter.java
```

文件 BluetoothAdapter.java 实现了所有蓝牙交互的入口。通过使用类 BluetoothAdapter，可以实现如下所示的功能：

- 发现其他的蓝牙设备，查询匹配的设备集。
- 使用一个已知蓝牙地址来初始化蓝牙设备 BluetoothDevice。
- 创建一个能够监听其他设备通信的类 BluetoothSocket。

(2)　请求远程蓝牙设备

本功能也不是由 Broadcom 公司提供的，而是由 Android SDK 提供的，源码位于如下所示的目录中：

```
framework/base/core/java/android.bluetooth/BluetoothDevice.java
```

文件 BluetoothDevice.java 代表一个远程蓝牙设备，可以支持 BLE 低功耗设备、BR/EDR 设备或 Dual-mode 类型的设备。通过使用类 BluetoothDevice，可以实现如下所示的功能：

- 请求获取远程蓝牙设备的连接。
- 查询获取远程蓝牙设备的名称、地址、类和链接状态。

(3)　实现客户端的低功耗蓝牙规范

在 Broadcom 公司提供的源码中，文件 BleClientProfile.java 的功能是实现客户端的低功耗蓝牙规范。在应用中要想访问远程设备中的低功耗蓝牙规范，就必须继承于类 BleClientProfile，并且需要提供要访问规范的必须参数和服务标识。通过 BleClientProfile 的派生类，可以发起一个远程设备的连接，并且一个 BleClientProfile 类可能会包含多个 BleClientService 对象的实例。

(4)　创建一个代表客户端角色设备上的低功耗蓝牙服务派生类

在 Broadcom 公司提供的源码中，文件 BleClientService.java 的功能是定义一个派生类，此派生类代表了客户端角色设备上的低功耗蓝牙服务。通过这个派生类，可以允许应用程序读写低功耗蓝牙服务的特征，并在特征改变时注册通知。

(5)　定义服务器端的角色的低功耗规范

在 Broadcom 公司提供的源码中，文件 BleServerProfile.java 的功能是定义了服务器端的角色的低功耗规范，在创建一个新的低功耗规范之前，需要先继承于这个类，并提供标识要访问

规范所必需的参数和服务。通常来说，一个 BleServerProfile 派生的类包含一个或多个 BleServerService 对象。在 BleServerProfile 派生的类中，包含低功耗规范中定义服务的 BleServerService 对象的集合。

(6) 创建低功耗服务

在 Broadcom 公司提供的源码中，文件 BleServerService.java 的功能是创建一个低功耗服务，这是服务器端角色上的低功耗规范的一部分。在 BleServerService 的派生类包含了一个或多个 BleCharacteristic 对象。在应用程序中，需要重写类 BleServerService 来实现一个服务。

(7) 描述低功耗蓝牙服务的特性

在 Broadcom 公司提供的源码中，文件 BleCharacteristic.java 的功能是描述低功耗蓝牙服务的特性。在特性中包含了描述符、实际值和元数据，提供了表现格式或便于阅读值的描述。

(8) 低功耗描述符

在 Broadcom 公司提供的源码中，文件 BleDescriptor.java 是 BleCharacteristic 的一部分，功能是定义一个低功耗描述符。

(9) 标识低功耗蓝牙规范、服务和特性

在 Broadcom 公司提供的源码中，文件 BleGattID.java 的功能是定义一个标识低功耗蓝牙规范、服务和特性的类，此类使用 16 位或 128 位的 UUIDs 来标识一个给定的低功耗蓝牙实体，这个实体包含规范、服务和特性。

(10) 为远程蓝牙设备提供额外信息

在 Broadcom 公司提供的源码中，文件 BleAdapter.java 的功能是为远程蓝牙设备提供额外的信息，能够判断远程设备是否是低功耗设备、BR/EDR 传统蓝牙设备或双模设备(同时支持低功耗和传统设备)。

(11) 保存与 GATT 相关的常量

在 Broadcom 公司提供的源码中，文件 BleConstants.java 的功能是定义保存各种与 GATT 相关的常量，这些常量用于表示各种实现低功耗功能函数的属性和返回值。

14.5 Android中的BlueDroid

了解了 Android 系统中低功耗蓝牙协议栈 BlueDroid 的基本知识后，在本节的内容中，将开始详细讲解 Android 源码中低功耗协议栈 BlueDroid 的具体实现和应用方法，为读者步入本书后面知识的学习打下基础。

14.5.1 Android系统中BlueDroid的架构

在 Android 新系统中，采用 BlueDroid 作为默认的协议栈，BlueDroid 分为如下所示的两个部分。

- Bluetooth Embedded System(BTE)：实现了 BT 的核心功能。
- Bluetooth Application Layer(BTA)：用于同 Android Framework 层进行交互。

在 Android 新系统中，BT 系统服务通过 JNI 与 BT Stack 进行交互，并且通过 Binder IPC 通信与应用交互，这个系统服务同时也提供给 RD，获取不同的 BT profiles。

图 14-4 展示了 BT Stack 的一个大体的结构。

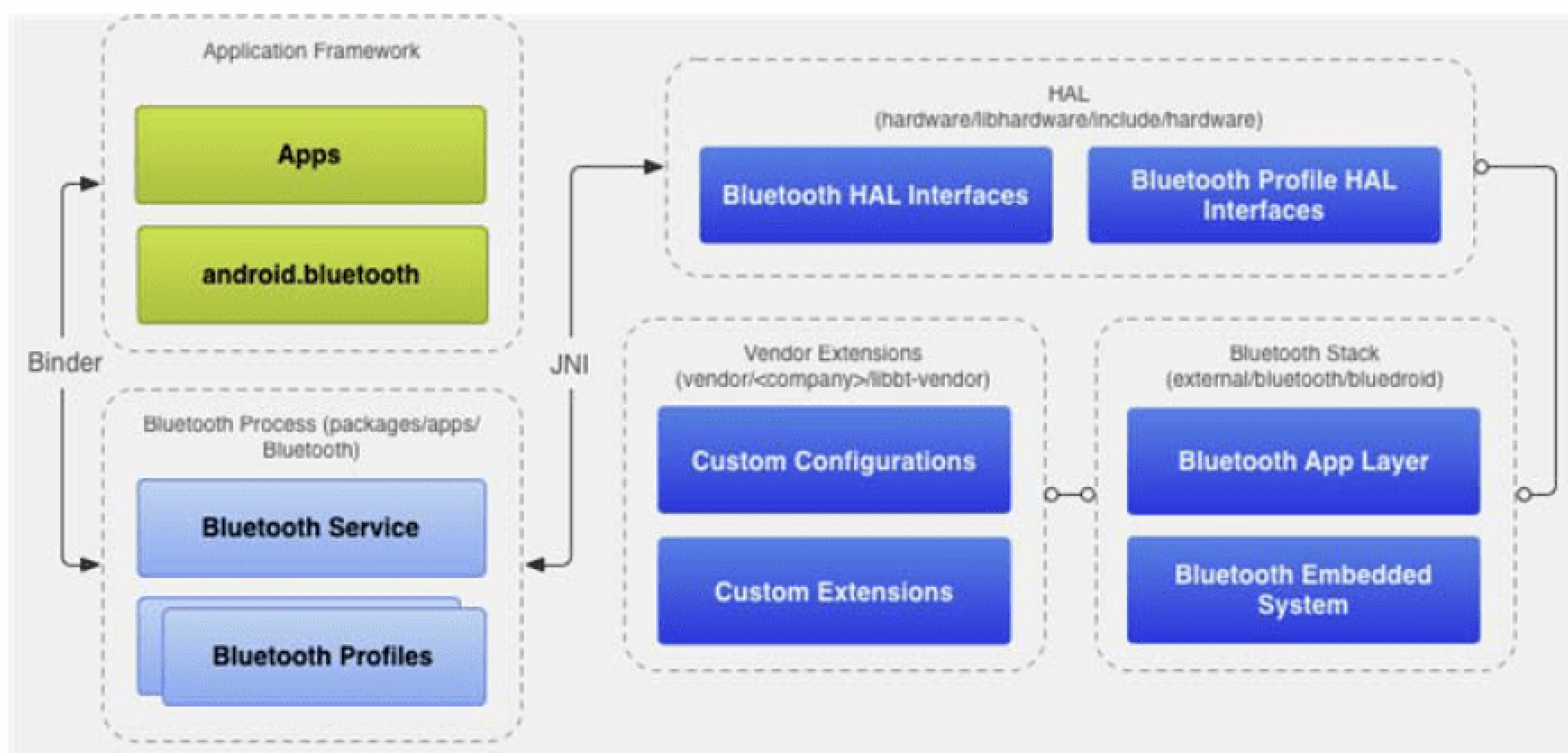


图 14-4　BT Stack的结构

14.5.2　Application Framework层分析

在 Application Framework 层中，功能是利用 android.bluetooth APIS 与 Bluetooth Hardware 层进行交互的，也就是通过 Binder IPC 机制调用 Bluetooth 进程。Application Framework 层的代码位于如下所示的目录中：

```
framework/base/core/java/android.bluetooth/
```

在文件 framework/base/core/java/android/bluetooth/BluetoothA2dp.java 之中，定义了 connect (Bluetoothevice)方法，功能是调用 Binder IPC 通信机制调用如下文件中的一个内部私有类：

```
packages/apps/Bluetooth/src/com/android/bluetooth/a2dp/A2dpService.java
```

文件 BluetoothA2dp.java 的具体实现代码如下所示：

```
public final class BluetoothA2dp implements BluetoothProfile {
    private static final String TAG = "BluetoothA2dp";
    private static final boolean DBG = true;
    private static final boolean VDBG = false;
    @SdkConstant(SdkConstantType.BROADCAST_INTENT_ACTION)
    public static final String ACTION_CONNECTION_STATE_CHANGED =
      "android.bluetooth.a2dp.profile.action.CONNECTION_STATE_CHANGED";
    @SdkConstant(SdkConstantType.BROADCAST_INTENT_ACTION)
    public static final String ACTION_PLAYING_STATE_CHANGED =
      "android.bluetooth.a2dp.profile.action.PLAYING_STATE_CHANGED";

    /**
     * A2DP sink device is streaming music. This state can be one of
     * {@link #EXTRA_STATE} or {@link #EXTRA_PREVIOUS_STATE} of
     * {@link #ACTION PLAYING STATE CHANGED} intent.
    public static final int STATE_PLAYING   =  10;
    public static final int STATE_NOT_PLAYING   =  11;
```

```
    private Context mContext;
    private ServiceListener mServiceListener;
    private IBluetoothA2dp mService;
    private BluetoothAdapter mAdapter;

    final private IBluetoothStateChangeCallback mBluetoothStateChangeCallback =
          new IBluetoothStateChangeCallback.Stub() {
             public void onBluetoothStateChange(boolean up) {
                if (DBG) Log.d(TAG, "onBluetoothStateChange: up=" + up);
                if (!up) {
                   if (VDBG) Log.d(TAG,"Unbinding service...");
                   synchronized (mConnection) {
                      try {
                         mService = null;
                         mContext.unbindService(mConnection);
                      } catch (Exception re) {
                         Log.e(TAG,"",re);
                      }
                   }
                } else {
                   synchronized (mConnection) {
                      try {
                         if (mService == null) {
                            if (VDBG) Log.d(TAG,"Binding service...");
                            if (!mContext.bindService(new Intent(
                               IBluetoothA2dp.class.getName()),
                               mConnection, 0)) {
                                Log.e(TAG, "Could not bind to Bluetooth A2DP Service");
                            }
                         }
                      } catch (Exception re) {
                         Log.e(TAG,"",re);
                      }
                   }
                }
             }
      };
    BluetoothA2dp(Context context, ServiceListener l) {
       mContext = context;
       mServiceListener = l;
       mAdapter = BluetoothAdapter.getDefaultAdapter();
       IBluetoothManager mgr = mAdapter.getBluetoothManager();
       if (mgr != null) {
          try {
             mgr.registerStateChangeCallback(mBluetoothStateChangeCallback);
          } catch (RemoteException e) {
             Log.e(TAG,"",e);
          }
       }
```

```
        if (!context.bindService(new Intent(IBluetoothA2dp.class.getName()),
          mConnection, 0)) {
            Log.e(TAG, "Could not bind to Bluetooth A2DP Service");
        }
    }

    void close() {
        mServiceListener = null;
        IBluetoothManager mgr = mAdapter.getBluetoothManager();
        if (mgr != null) {
            try {
                mgr.unregisterStateChangeCallback(mBluetoothStateChangeCallback);
            } catch (Exception e) {
                Log.e(TAG,"",e);
            }
        }
        synchronized (mConnection) {
            if (mService != null) {
                try {
                    mService = null;
                    mContext.unbindService(mConnection);
                } catch (Exception re) {
                    Log.e(TAG,"",re);
                }
            }
        }
    }
    public void finalize() {
        close();
    }
    public boolean connect(BluetoothDevice device) {
        if (DBG) log("connect(" + device + ")");
        if (mService != null && isEnabled()
          && isValidDevice(device)) {
            try {
                return mService.connect(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }
    public boolean disconnect(BluetoothDevice device) {
        if (DBG) log("disconnect(" + device + ")");
        if (mService != null && isEnabled() &&
            isValidDevice(device)) {
```

```
            try {
                return mService.disconnect(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }

    public List<BluetoothDevice> getConnectedDevices() {
        if (VDBG) log("getConnectedDevices()");
        if (mService != null && isEnabled()) {
            try {
                return mService.getConnectedDevices();
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return new ArrayList<BluetoothDevice>();
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return new ArrayList<BluetoothDevice>();
    }
    public List<BluetoothDevice> getDevicesMatchingConnectionStates(int[] states) {
        if (VDBG) log("getDevicesMatchingStates()");
        if (mService != null && isEnabled()) {
            try {
                return mService.getDevicesMatchingConnectionStates(states);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return new ArrayList<BluetoothDevice>();
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return new ArrayList<BluetoothDevice>();
    }
    public int getConnectionState(BluetoothDevice device) {
        if (VDBG) log("getState(" + device + ")");
        if (mService != null && isEnabled()
          && isValidDevice(device)) {
            try {
                return mService.getConnectionState(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return BluetoothProfile.STATE_DISCONNECTED;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
```

```
        return BluetoothProfile.STATE_DISCONNECTED;
    }
    public boolean setPriority(BluetoothDevice device, int priority) {
        if (DBG) log("setPriority(" + device + ", " + priority + ")");
        if (mService != null && isEnabled()
          && isValidDevice(device)) {
            if (priority != BluetoothProfile.PRIORITY_OFF &&
                priority != BluetoothProfile.PRIORITY_ON){
              return false;
            }
            try {
                return mService.setPriority(device, priority);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }
    public int getPriority(BluetoothDevice device) {
        if (VDBG) log("getPriority(" + device + ")");
        if (mService != null && isEnabled()
          && isValidDevice(device)) {
            try {
                return mService.getPriority(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return BluetoothProfile.PRIORITY OFF;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return BluetoothProfile.PRIORITY_OFF;
    }
    public boolean isA2dpPlaying(BluetoothDevice device) {
        if (mService != null && isEnabled()
          && isValidDevice(device)) {
            try {
                return mService.isA2dpPlaying(device);
            } catch (RemoteException e) {
                Log.e(TAG, "Stack:" + Log.getStackTraceString(new Throwable()));
                return false;
            }
        }
        if (mService == null) Log.w(TAG, "Proxy not attached to service");
        return false;
    }
    public boolean shouldSendVolumeKeys(BluetoothDevice device) {
        if (isEnabled() && isValidDevice(device)) {
```

```
        ParcelUuid[] uuids = device.getUuids();
        if (uuids == null) return false;

        for (ParcelUuid uuid: uuids) {
            if (BluetoothUuid.isAvrcpTarget(uuid)) {
                return true;
            }
        }
    }
    return false;
}
public static String stateToString(int state) {
    switch (state) {
    case STATE_DISCONNECTED:
        return "disconnected";
    case STATE_CONNECTING:
        return "connecting";
    case STATE_CONNECTED:
        return "connected";
    case STATE_DISCONNECTING:
        return "disconnecting";
    case STATE_PLAYING:
        return "playing";
    case STATE_NOT_PLAYING:
      return "not playing";
    default:
        return "<unknown state " + state + ">";
    }
}

private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        if (DBG) Log.d(TAG, "Proxy object connected");
        mService = IBluetoothA2dp.Stub.asInterface(service);

        if (mServiceListener != null) {
            mServiceListener.onServiceConnected(
              BluetoothProfile.A2DP, BluetoothA2dp.this);
        }
    }
    public void onServiceDisconnected(ComponentName className) {
        if (DBG) Log.d(TAG, "Proxy object disconnected");
        mService = null;
        if (mServiceListener != null) {
            mServiceListener.onServiceDisconnected(BluetoothProfile.A2DP);
        }
    }
};
}
```

上述代码中定义了 A2dpService 对象 service，并调用了 getService()方法。A2dpService 是一个继承于类 ProfileService 的子类，而 ProfileService 是继承于类 Service 的子类。文件 A2dpService.java 的主要实现代码如下所示：

```
public class A2dpService extends ProfileService {
    private static final boolean DBG = false;
    private static final String TAG="A2dpService";

    private A2dpStateMachine mStateMachine;
    private Avrcp mAvrcp;
    private static A2dpService sAd2dpService;

    protected String getName() {
        return TAG;
    }

    protected IProfileServiceBinder initBinder() {
        return new BluetoothA2dpBinder(this);
    }

    protected boolean start() {
        mStateMachine = A2dpStateMachine.make(this, this);
        mAvrcp = Avrcp.make(this);
        setA2dpService(this);
        return true;
    }

    protected boolean stop() {
        mStateMachine.doQuit();
        mAvrcp.doQuit();
        return true;
    }

    protected boolean cleanup() {
        if (mStateMachine!= null) {
            mStateMachine.cleanup();
        }
        if (mAvrcp != null) {
            mAvrcp.cleanup();
            mAvrcp = null;
        }
        clearA2dpService();
        return true;
    }

    //API Methods

    public static synchronized A2dpService getA2dpService(){
```

```
        if (sAd2dpService != null && sAd2dpService.isAvailable()) {
            if (DBG) Log.d(TAG, "getA2DPService(): returning " + sAd2dpService);
            return sAd2dpService;
        }
        if (DBG)  {
            if (sAd2dpService == null) {
                Log.d(TAG, "getA2dpService(): service is NULL");
            } else if (!(sAd2dpService.isAvailable())) {
                Log.d(TAG,"getA2dpService(): service is not available");
            }
        }
        return null;
    }

    private static synchronized void setA2dpService(A2dpService instance) {
        if (instance != null && instance.isAvailable()) {
            if (DBG) Log.d(TAG, "setA2dpService(): set to: " + sAd2dpService);
            sAd2dpService = instance;
        } else {
            if (DBG)  {
                if (sAd2dpService == null) {
                    Log.d(TAG, "setA2dpService(): service not available");
                } else if (!sAd2dpService.isAvailable()) {
                    Log.d(TAG,"setA2dpService(): service is cleaning up");
                }
            }
        }
    }

    private static synchronized void clearA2dpService() {
        sAd2dpService = null;
    }

    public boolean connect(BluetoothDevice device) {
        enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
                                       "Need BLUETOOTH ADMIN permission");

        if (getPriority(device) == BluetoothProfile.PRIORITY_OFF) {
            return false;
        }

        int connectionState = mStateMachine.getConnectionState(device);
        if (connectionState == BluetoothProfile.STATE_CONNECTED
          || connectionState == BluetoothProfile.STATE_CONNECTING) {
            return false;
        }

        mStateMachine.sendMessage(A2dpStateMachine.CONNECT, device);
        return true;
```

```
    }

    boolean disconnect(BluetoothDevice device) {
        enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
                                       "Need BLUETOOTH ADMIN permission");
        int connectionState = mStateMachine.getConnectionState(device);
        if (connectionState != BluetoothProfile.STATE_CONNECTED
          && connectionState != BluetoothProfile.STATE_CONNECTING) {
            return false;
        }

        mStateMachine.sendMessage(A2dpStateMachine.DISCONNECT, device);
        return true;
    }
}
```

由此可见，在接下来的通信过程中，通过 Binder IPC 通信机制调用了文件 A2dpService.java 中的 connect(BluetoothDevice)方法。上述过程就是 Bluetooth Application Framework 与 Bluetooth Process 之间的调用过程。

14.5.3　分析Bluetooth System Service层

在 Android 系统中，Bluetooth System Service 位于 packages/apps/Bluetooth 目录下，将其打包成一个“Android App(Android 应用程序)”包，并且在 Android Framework 层实现 BT Service 和各种 profile。BT App 接下来会通过 JNI 调用到 HAL 层。

在文件 A2dpService.java 中，connect 方法会发送一个 StateMachine.sendMessage (A2dpStateMachine.CONNECT, device)的 message 信息，这个 message 会被 A2dpStateMachine 对象的 processMessage(Message)方法接收到，对应代码如下所示：

```
case CONNECT:
    BluetoothDevice device = (BluetoothDevice)message.obj;
    broadcastConnectionState(device, BluetoothProfile.STATE_CONNECTING,
                             BluetoothProfile.STATE_DISCONNECTED);

    if (!connectA2dpNative(getByteAddress(device))) {
        broadcastConnectionState(device, BluetoothProfile.STATE_DISCONNECTED,
                                 BluetoothProfile.STATE_CONNECTING);
        break;
    }
    synchronized (A2dpStateMachine.this) {
        mTargetDevice = device;
        transitionTo(mPending);
    }
    // TODO(BT) remove CONNECT_TIMEOUT when the stack
    //          sends back events consistently
    sendMessageDelayed(CONNECT_TIMEOUT, 30000);
    break;
```

在上述代码中，会通过“connectA2dpNative(getByteAddress(device));”代码行设置通过 JNI 调用到 Native(本地程序)：

```
private native boolean connectA2dpNative(byte[] address);
```

14.5.4　分析JNI层

在 Android 系统中，与 Bluetooth 有关的 JNI 代码位于如下所示的目录中：

```
packages/apps/bluetooth/jni
```

JNI 层的代码会调用到 HAL 层，并且在确信一些 BT 操作被触发时从 HAL 获取一些回调，例如当 BT 设备被发现时。例如在 A2dp 连接的例子中，BT System Service 会通过 JNI 调用文件 com_android_bluetooth_a2dp.cpp 中的方法，此文件的主要实现代码如下所示：

```
namespace android {
    static jmethodID method_onConnectionStateChanged;
    static jmethodID method_onAudioStateChanged;

    static const btav_interface_t *sBluetoothA2dpInterface = NULL;
    static jobject mCallbacksObj = NULL;
    static JNIEnv *sCallbackEnv = NULL;

    static bool checkCallbackThread() {
        sCallbackEnv = getCallbackEnv();
    }

    JNIEnv *env = AndroidRuntime::getJNIEnv();
    if (sCallbackEnv!=env || sCallbackEnv==NULL) return false;
    return true;
}

static void bta2dp_connection_state_callback(btav_connection_state_t state,
  bt_bdaddr_t *bd_addr) {
    jbyteArray addr;

    ALOGI("%s", __FUNCTION__);

    if (!checkCallbackThread()) {
        ALOGE("Callback: '%s' is not called on the correct thread", __FUNCTION__);
        return;
    }
    addr = sCallbackEnv->NewByteArray(sizeof(bt bdaddr t));
    if (!addr) {
        ALOGE("Fail to new jbyteArray bd addr for connection state");
        checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
        return;
    }
```

```
    sCallbackEnv->SetByteArrayRegion(addr, 0,
      sizeof(bt bdaddr t), (jbyte*)bd addr);
    sCallbackEnv->CallVoidMethod(mCallbacksObj,
      method_onConnectionStateChanged, (jint)state, addr);
    checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
    sCallbackEnv->DeleteLocalRef(addr);
}

static void bta2dp_audio_state_callback(btav_audio_state_t state,
  bt_bdaddr_t *bd_addr) {
    jbyteArray addr;

    ALOGI("%s", __FUNCTION__);

    if (!checkCallbackThread()) {
    ALOGE("Callback: '%s' is not called on the correct thread", __FUNCTION__);
        return;
    }
    addr = sCallbackEnv->NewByteArray(sizeof(bt_bdaddr_t));
    if (!addr) {
        ALOGE("Fail to new jbyteArray bd addr for connection state");
        checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
        return;
    }

    sCallbackEnv->SetByteArrayRegion(addr, 0,
      sizeof(bt_bdaddr_t), (jbyte*)bd_addr);
    sCallbackEnv->CallVoidMethod(mCallbacksObj, method_onAudioStateChanged,
      (jint)state, addr);
    checkAndClearExceptionFromCallback(sCallbackEnv, __FUNCTION__);
    sCallbackEnv->DeleteLocalRef(addr);
}

static btav_callbacks_t sBluetoothA2dpCallbacks = {
    sizeof(sBluetoothA2dpCallbacks),
    bta2dp_connection_state_callback,
    bta2dp_audio_state_callback
};

static void classInitNative(JNIEnv *env, jclass clazz) {
    int err;
    const bt_interface_t *btInf;
    bt_status_t status;

    method_onConnectionStateChanged =
      env->GetMethodID(clazz, "onConnectionStateChanged", "(I[B)V");

    method onAudioStateChanged =
      env->GetMethodID(clazz, "onAudioStateChanged", "(I[B)V");
```

```
    if ((btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if ((sBluetoothA2dpInterface = (btav_interface_t*)
      btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
        ALOGE("Failed to get Bluetooth A2DP Interface");
        return;
    }
    /*
    if ( (status = sBluetoothA2dpInterface->init(&sBluetoothA2dpCallbacks))
      != BT_STATUS_SUCCESS) {
        ALOGE("Failed to initialize Bluetooth A2DP, status: %d", status);
        sBluetoothA2dpInterface = NULL;
        return;
    }*/

    ALOGI("%s: succeeds", __FUNCTION__);
}

static void initNative(JNIEnv *env, jobject object) {
    const bt_interface_t* btInf;
    bt_status_t status;

    if ( (btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if (sBluetoothA2dpInterface != NULL) {
        ALOGW("Cleaning up A2DP Interface before initializing...");
        sBluetoothA2dpInterface->cleanup();
        sBluetoothA2dpInterface = NULL;
    }

    if (mCallbacksObj != NULL) {
        ALOGW("Cleaning up A2DP callback object");
        env->DeleteGlobalRef(mCallbacksObj);
        mCallbacksObj = NULL;
    }

    if ((sBluetoothA2dpInterface = (btav_interface_t*)
      btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
        ALOGE("Failed to get Bluetooth A2DP Interface");
        return;
    }
```

```
    if ((status = sBluetoothA2dpInterface->init(&sBluetoothA2dpCallbacks))
      != BT_STATUS_SUCCESS) {
        ALOGE("Failed to initialize Bluetooth A2DP, status: %d", status);
        sBluetoothA2dpInterface = NULL;
        return;
    }
    mCallbacksObj = env->NewGlobalRef(object);
}
static void cleanupNative(JNIEnv *env, jobject object) {
    const bt_interface_t *btInf;
    bt_status_t status;

    if ((btInf = getBluetoothInterface()) == NULL) {
        ALOGE("Bluetooth module is not loaded");
        return;
    }

    if (sBluetoothA2dpInterface !=NULL) {
        sBluetoothA2dpInterface->cleanup();
        sBluetoothA2dpInterface = NULL;
    }

    if (mCallbacksObj != NULL) {
        env->DeleteGlobalRef(mCallbacksObj);
        mCallbacksObj = NULL;
    }
}

static jboolean connectA2dpNative(JNIEnv *env, jobject object, jbyteArray address)
{
    jbyte *addr;
    bt_bdaddr_t * btAddr;
    bt_status_t status;

    ALOGI("%s: sBluetoothA2dpInterface: %p", __FUNCTION__,
      sBluetoothA2dpInterface);
    if (!sBluetoothA2dpInterface) return JNI_FALSE;

    addr = env->GetByteArrayElements(address, NULL);
    btAddr = (bt_bdaddr_t*)addr;
    if (!addr) {
        jniThrowIOException(env, EINVAL);
        return JNI_FALSE;
    }

    if ((status = sBluetoothA2dpInterface->connect((bt_bdaddr_t*)addr))
      != BT_STATUS_SUCCESS) {
        ALOGE("Failed HF connection, status: %d", status);
    }
```

```
    env->ReleaseByteArrayElements(address, addr, 0);
    return (status == BT STATUS SUCCESS)? JNI TRUE : JNI FALSE;
}

static jboolean disconnectA2dpNative(JNIEnv *env, jobject object,
  jbyteArray address) {
    jbyte *addr;
    bt_status_t status;

    if (!sBluetoothA2dpInterface) return JNI_FALSE;

    addr = env->GetByteArrayElements(address, NULL);
    if (!addr) {
        jniThrowIOException(env, EINVAL);
        return JNI_FALSE;
    }

    if ((status = sBluetoothA2dpInterface->disconnect((bt_bdaddr_t*)addr))
      != BT_STATUS_SUCCESS) {
        ALOGE("Failed HF disconnection, status: %d", status);
    }
    env->ReleaseByteArrayElements(address, addr, 0);
    return (status==BT_STATUS_SUCCESS)? JNI_TRUE : JNI_FALSE;
}

static JNINativeMethod sMethods[] = {
    {"classInitNative", "()V", (void*)classInitNative},
    {"initNative", "()V", (void*)initNative},
    {"cleanupNative", "()V", (void*)cleanupNative},
    {"connectA2dpNative", "([B)Z", (void*)connectA2dpNative},
    {"disconnectA2dpNative", "([B)Z", (void*)disconnectA2dpNative},
};

int register_com_android_bluetooth_a2dp(JNIEnv *env)
{
    return jniRegisterNativeMethods(env,
      "com/android/bluetooth/a2dp/A2dpStateMachine",
      sMethods, NELEM(sMethods));
}
}
```

在上述代码中用到了结构体对象 sBluetoothA2dpInterface，此对象在方法 initNative(JNIEnv *env, jobject object)中获取，即如下所示的代码：

```
if ((sBluetoothA2dpInterface = (btav_interface_t*)
  btInf->get_profile_interface(BT_PROFILE_ADVANCED_AUDIO_ID)) == NULL) {
    ALOGE("Failed to get Bluetooth A2DP Interface");
    return;
}
```

14.5.5　分析HAL层

硬件抽象层用于定义 android.bluetooth APIs 和 BT process 调用的标准接口，BT HAL 的头文件位于如下所示的文件中：

```
hardware/libhardware/include/hardware/bluetooth.h
hardware/libhardware/include/hardware/bt_*.h
```

JNI 中 sBluetoothA2dpInterface 是一个 btav_interface_t 结构体，位于 hardware/libhardware/include/hardware/bt_av.h 中，具体定义代码如下所示：

```
typedef struct {
    size_t  size;
    bt_status_t  (*init)(btav_callbacks_t *callbacks);
    bt_status_t  (*connect)(bt_bdaddr_t *bd_addr);
    bt_status_t  (*disconnect)(bt_bdaddr_t *bd_addr);
    void  (*cleanup)(void);
} btav_interface_t;
```

Android 系统新版本默认蓝牙协议栈 BlueDroid 在如下所示的目录下实现：

```
external/bluetooth/bluedroid
```

上述 stack 实现了通用的 BT HAL 并且也可以通过扩展和改变配置来自定义。例如 A2dp 的连接会调用到 external/bluetooth/bluedroid/btif/src/btif_av.c 的 connect 方法，此方法的具体实现代码如下所示：

```
static bt_status_t connect(bt_bdaddr_t *bd_addr)
{
    BTIF_TRACE_EVENT1("%s", __FUNCTION__);
    CHECK_BTAV_INIT();

    return btif_queue_connect(UUID_SERVCLASS_AUDIO_SOURCE, bd_addr, connect_int);
}
```

14.6　Android蓝牙模块的运作流程

经过本章前面内容的学习，已经了解了 Android 系统中蓝牙模块内核代码的基本知识。在本节的内容中，将以 Android 源码为蓝本，介绍 Android 4.3 系统中蓝牙模块的具体运作流程。

14.6.1　打开蓝牙设备

在 Android 系统的内置蓝牙模块中，打开蓝牙功能的开关是 settings 选项中的 switch 开关。打开后，需要使用 systemServer.java 的代码来开启蓝牙服务。

相关的代码如下所示：

```
if (SystemProperties.get("ro.kernel.qemu").equals("1")) {
    Slog.i(TAG, "No Bluetooh Service (emulator)");
} else if (factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL) {
    Slog.i(TAG, "No Bluetooth Service (factory test)");
} else {
    Slog.i(TAG, "Bluetooth Manager Service");
    bluetooth = new BluetoothManagerService(context);
    ServiceManager.addService(
      BluetoothAdapter.BLUETOOTH_MANAGER_ SERVICE, bluetooth);
}
```

在类 bluetoothManagerService 的构造方中，方法 loadStoredNameAndAddress()用于读取蓝牙并打开默认的名称，方法 isBluetoothPersistedStateOn()用于判断是否已经打开蓝牙，如果已经打开，则后面的操作需要执行开启蓝牙的动作，前面的注册广播代码就起了这个作用。

对应的代码如下所示：

```
BluetoothManagerService(Context context) {
    //...一些变量声明初始化...
    IntentFilter filter = new IntentFilter(Intent.ACTION_BOOT_COMPLETED);
    filter.addAction(BluetoothAdapter.ACTION_LOCAL_NAME_CHANGED);
    filter.addAction(Intent.ACTION_USER_SWITCHED);
    registerForAirplaneMode(filter);
    mContext.registerReceiver(mReceiver, filter);
    loadStoredNameAndAddress();
    if (isBluetoothPersistedStateOn()) {
        mEnableExternal = true;
    }
}
```

返回到开关界面，界面开关实现文件是 BluetoothEnabler.java，而方法 setBluetoothEnabled()是界面开关的具体实现，对应代码如下所示：

```
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    // Show toast message if Bluetooth is not allowed in airplane mode
    if (isChecked && !WirelessSettings.isRadioAllowed(
      mContext, Settings.Global.RADIO_BLUETOOTH)) {
        Toast.makeText(mContext, R.string.wifi_in_airplane_mode,
          Toast.LENGTH_SHORT).show();
        // Reset switch to off
        buttonView.setChecked(false);
    }

    if (mLocalAdapter != null) {
        mLocalAdapter.setBluetoothEnabled(isChecked);
    }
    mSwitch.setEnabled(false);
}
```

在上述代码中，判断了是否是飞行模式，如果是飞行模式，则会弹出 toast 提示，通过阅

读方法 setBluetoothEnabled()可知，mLocalAdapter(LocalBluetoothAdapter)只是个过渡方法，里面的 mAdapter(BluetoothAdapter)方法才是真正的主角，对应代码如下所示：

```
public void setBluetoothEnabled(boolean enabled) {
    boolean success = enabled? mAdapter.enable() : mAdapter.disable();

    if (success) {
        setBluetoothStateInt(enabled?
        BluetoothAdapter.STATE_TURNING_ON : BluetoothAdapter.STATE_TURNING_OFF);
    } else {
        //...
    }
}
```

文件 BluetoothAdapter.java 实现了一个单例模式的应用，提供了供其他程序调用蓝牙的一些方法，外部程序在调用蓝牙方法之前需要用到文件 BluetoothAdapter.java，此文件中的 BluetoothAdapter 对象演示了 Android 系统典型的 binder 应用。对应代码如下所示：

```
public static synchronized BluetoothAdapter getDefaultAdapter() {
    if (sAdapter == null) {
        IBinder b = ServiceManager.getService(BLUETOOTH_MANAGER_SERVICE);
        if (b != null) {
            IBluetoothManager managerService =
             IBluetoothManager.Stub.asInterface(b);
            sAdapter = new BluetoothAdapter(managerService);
        } else {
            Log.e(TAG, "Bluetooth binder is null");
        }
    }
    return sAdapter;
}
```

调用 mAdapter.enable()方法之后，外部其他应用也需要调用 enable()方法。

这里的文件 BluetoothAdapter 出于 Framework 层，文件 BluetoothAdapter.java 中的 enable()方法调用会先回到文件 BluetoothManagerService.java 中的 enable()方法，然后来到文件 BluetoothManagerService.java 中的 handleEnable()方法，后面需要跳转到新类：

```
private void handleEnable(boolean persist, boolean quietMode) {
     synchronized(mConnection) {
        if ((mBluetooth == null) && (!mBinding)) {
            //Start bind timeout and bind
            Message timeoutMsg = mHandler.obtainMessage(MESSAGE_TIMEOUT_BIND);
            mHandler.sendMessageDelayed(timeoutMsg,TIMEOUT_BIND_MS);
            mConnection.setGetNameAddressOnly(false);
            Intent i = new Intent(IBluetooth.class.getName());
            if (!mContext.bindService(i, mConnection,Context.BIND_AUTO_CREATE,
              UserHandle.USER_CURRENT)) {
                mHandler.removeMessages(MESSAGE_TIMEOUT_BIND);
                Log.e(TAG, "Fail to bind to: " + IBluetooth.class.getName());
```

```
            } else {
                mBinding = true;
            }
        }
        ...
```

分析如下所示的 log 信息：

```
ActivityManager: Start proc com.android.bluetooth for service
com.android.bluetooth/.btservice.AdapterService:"
```

在 AdapterService 服务中一共有三个 enable(),跳转关系非常简单,其中最后一个比较关键：

```
public synchronized boolean enable(boolean quietMode) {
    enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
                                   "Need BLUETOOTH ADMIN permission");
    if (DBG) debugLog("Enable called with quiet mode status =  " + mQuietmode);
    mQuietmode = quietMode;
    Message m = mAdapterStateMachine.obtainMessage(AdapterState.USER_TURN_ON);
    mAdapterStateMachine.sendMessage(m);
    return true;
}
```

这样将从一个状态接受命令跳到另一个状态，因为是在开启蓝牙，所以先去文件 AdapterState.java 的内部类 offstate.java 中找，在这个分支的 USER_TURN_ON 中会看到方法 mAdapterService.processStart()，在里面可以看到蓝牙遍历下所支持的 profile，最后会发出一个带 AdapterState.STARTED 标识的消息。具体处理功能在 AdapterState.java 中实现，对应代码如下所示：

```
case STARTED:   {
    if (DBG) Log.d(TAG,"CURRENT_STATE=PENDING, MESSAGE = STARTED, isTurningOn="
      + isTurningOn + ", isTurningOff=" + isTurningOff);
    //Remove start timeout
    removeMessages(START_TIMEOUT);

    //Enable
    boolean ret = mAdapterService.enableNative();
    if (!ret) {
        Log.e(TAG, "Error while turning Bluetooth On");
        notifyAdapterStateChange(BluetoothAdapter.STATE_OFF);
        transitionTo(mOffState);
    } else {
        sendMessageDelayed(ENABLE_TIMEOUT, ENABLE_TIMEOUT_DELAY);
    }
```

在上述代码中，使用 JNI 调用了 enableNative()函数，根据 Android JNI 的函数命名习惯，可以很容易找到函数 enableNative 对应的 C++函数，在如下文件中实现：

```
packages/apps/Bluetooth/jni/com_android_bluetooth_btservice_AdapterService.cpp
```

函数 enableNative 的具体实现代码如下所示：

```
static jboolean enableNative(JNIEnv *env, jobject obj) {
    ALOGV("%s:",__FUNCTION__);
    jboolean result = JNI_FALSE;
    if (!sBluetoothInterface) return result;
    int ret = sBluetoothInterface->enable();
    result = (ret==BT_STATUS_SUCCESS)? JNI_TRUE : JNI_FALSE;
    return result;
}
```

在上述代码中，sBluetoothInterface 在如下所示的文件中定义：

```
/external/bluetooth/bluedroid/btif/src/bluetooth.c
```

定义 bt_interface_t bluetoothInterface 的代码如下所示：

```
static const bt_interface_t bluetoothInterface = {
    sizeof(bt_interface_t),
    init,
    enable,
    disable,
    ...
    start_discovery,
    cancel_discovery,
    create_bond,
    remove_bond,
    cancel_bond,
    ...
};
```

函数 enable()在 bluetooth.c 中实现，对应的代码如下所示：

```
static int enable(void)
{
    ALOGI("enable");

    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;

    return btif_enable_bluetooth();
}
```

在上述代码中用到了函数 bt_status_t btif_enable_bluetooth，具体实现代码如下所示：

```
bt_status_t btif_enable_bluetooth(void)
{
    BTIF_TRACE_DEBUG0("BTIF ENABLE BLUETOOTH");

    if (btif_core_state != BTIF_CORE_STATE_DISABLED)
    {
        ALOGD("not disabled\n");
```

```
        return BT_STATUS_DONE;
    }

    btif_core_state = BTIF_CORE_STATE_ENABLING;

    /* Create the GKI tasks and run them */
    bte_main_enable(btif_local_bd_addr.address);

    return BT_STATUS_SUCCESS;
}
```

上述代码中调用了函数 bte_main_enable，此函数在文件 external/bluetooth/bluedroid/main/bte_main.c 中定义，具体实现代码如下所示：

```
void bte_main_enable(uint8_t *local_addr)
{
    APPL_TRACE_DEBUG1("%s", __FUNCTION__);
    ...

#if (defined (BT_CLEAN_TURN_ON_DISABLED) && BT_CLEAN_TURN_ON_DISABLED == TRUE)
    APPL_TRACE_DEBUG1("%s  Not Turninig Off the BT before Turninig ON",
                      __FUNCTION__);

#else
    /* toggle chip power to ensure we will reset chip in case
    a previous stack shutdown wasn't completed gracefully */
    bt_hc_if->set_power(BT_HC_CHIP_PWR_OFF);
#endif
    bt_hc_if->set_power(BT_HC_CHIP_PWR_ON);

    bt_hc_if->preload(NULL);
}
```

在上述代码中调用了文件 external/bluetooth/bluedroid/hci/src/bt_hci_bdroid.c 中的函数 set_power，具体实现代码如下所示：

```
static void set_power(bt_hc_chip_power_state_t state)
{
    int pwr_state;

    BTHCDBG("set_power %d", state);

    /* Calling vendor-specific part */
    pwr_state = (state==BT_HC_CHIP_PWR_ON) ? BT_VND_PWR_ON : BT_VND_PWR_OFF;

    if (bt_vnd_if)
        bt_vnd_if->op(BT_VND_OP_POWER_CTRL, &pwr_state);
    else
        ALOGE("vendor lib is missing!");
}
```

在上述代码中，bt_vnd_if 源于文件 external/bluetooth/bluedroid/hci/include/bt_vendor_lib.h，对应的代码如下所示：

```
extern const bt vendor interface t BLUETOOTH VENDOR LIB INTERFACE;
bt_vendor_interface_t *bt_vnd_if = NULL;
```

谷歌已经定义好了接口，具体实现需要由 Vendor 厂商来完成，这部分代码需要看各家芯片商的因缘，通常不会公开。打开蓝牙功能需要用如下类似的字符串实现：

```
static const char *BT_DRIVER_MODULE_PATH = "/system/lib/modules/mbt8xxx.ko";
static const char *BT DRIVER MODULE NAME = "bt8xxx";
static const char *BT_DRIVER_MODULE_INIT_ARG = " init_cfg=";
static const char *BT_DRIVER_MODULE_INIT_CFG_PATH = "bt_init_cfg.conf";
```

还有类似下面的动作，insmod 加载驱动，rfkill 控制上下电，不同厂商的具体做法会有所不同：

```
ret = insmod(BT DRIVER MODULE PATH, arg buf);
ret = system("/system/bin/rfkill block all");
```

到此为止，打开 Android 蓝牙的流程介绍全部结束。对于 Vendor 那部分的代码，需要看各自厂商的代码，通常在开启蓝牙后才会通电，这样比较符合逻辑，并节省电量。查看是否通电的方法是，连上手机，用 adb shell 看 sys/class/rfkill 目录下的 state 的状态值，有些厂商会把蓝牙和 Wi-Fi 的上电算在一起，这一点是需要特别注意的，避免误判。

14.6.2　搜索蓝牙

在 Android 系统中，有如下两种启动蓝牙搜索工作的形式：

- 在蓝牙设置界面开启蓝牙会直接开始搜索。
- 先打开蓝牙开关，在进入蓝牙设置界面时也会触发搜索。

上述两种方式在最后都要来到文件 BluetoothSettngs.java 中的方法 startScanning()，此方法的代码如下所示：

```
private void updateContent(int bluetoothState, boolean scanState) {
    if (numberOfPairedDevices == 0) {
        preferenceScreen.removePreference(mPairedDevicesCategory);
        if (scanState == true) {
            mActivityStarted = false;
            startScanning();
        } else
        ...
}
private void startScanning() {
    if (!mAvailableDevicesCategoryIsPresent) {
        getPreferenceScreen().addPreference(mAvailableDevicesCategory);
    }
    mLocalAdapter.startScanning(true);
}
```

可见蓝牙的搜索和打开流程在结构上是一致的，需要利用文件 LocalBluetoothAdapter.java 过渡到文件 BluetoothAdapter.java，然后再跳转至文件 AdapterService.java。在上述过渡过程中，startScaning()方法变成了 startDiscovery()方法，startScaning()方法在如下所示的文件中实现：

```
packages/apps/Settings/src/com/android/settings/bluetooth/LocalBluetoothAdapter.java
```

startScaning()方法的实现代码如下所示：

```
void startScanning(boolean force) {
    if (!mAdapter.isDiscovering()) {
        if (!force) {
            // Don't scan more than frequently than SCAN_EXPIRATION_MS,
            // unless forced
            if (mLastScan + SCAN_EXPIRATION_MS > System.currentTimeMillis()) {
                return;
            }
            // If we are playing music, don't scan unless forced.
            A2dpProfile a2dp = mProfileManager.getA2dpProfile();
            if (a2dp!=null && a2dp.isA2dpPlaying()) {
                return;
            }
        }
        //这里才是我们最关注的，前面的限制条件关注一下就行了
        if (mAdapter.startDiscovery()) {
            mLastScan = System.currentTimeMillis();
        }
    }
}
```

方法 startDiscovery()在文件 frameworks/base/core/java/android/bluetooth/BluetoothAdapter.java 中定义，具体代码如下所示：

```
public boolean startDiscovery() {
    ...
    AdapterService service = getService();
    if (service == null) return false;
    return service.startDiscovery();
}
```

在上述代码中，service.startDiscovery()在如下所示的文件中定义：

```
packages/apps/Bluetooth/src/com/android/bluetooth/btservice/AdapterService.java
```

方法 service.startDiscovery()的具体实现代码如下所示：

```
boolean startDiscovery() {
    enforceCallingOrSelfPermission(BLUETOOTH_ADMIN_PERM,
            "Need BLUETOOTH ADMIN permission");

    return startDiscoveryNative();
}
```

JNI 文件 packages/apps/Bluetooth/jni/com_android_bluetooth_btservice_AdapterService.cpp 的对应代码如下所示：

```
static jboolean startDiscoveryNative(JNIEnv *env, jobject obj) {
    ALOGV("%s:",__FUNCTION__);

    jboolean result = JNI_FALSE;
    if (!sBluetoothInterface) return result;

    int ret = sBluetoothInterface->start_discovery();
    result = (ret==BT_STATUS_SUCCESS)? JNI_TRUE : JNI_FALSE;
    return result;
}
```

在上述代码中，函数 start_discovery 在文件 external/bluetooth/bluedroid/btif/src/bluetooth.c 中定义，具体实现代码如下所示：

```
static int start_discovery(void)
{
    /* sanity check */
    if (interface_ready() == FALSE)
        return BT_STATUS_NOT_READY;

    return btif_dm_start_discovery();
}
```

在上述代码中，函数 btif_dm_start_discovery 在文件 external/bluetooth/bluedroid/btif/src/btif_dm.c 中定义，具体实现代码如下所示：

```
bt_status_t  btif_dm_start_discovery(void)
{
    tBTA_DM_INQ inq_params;
    tBTA_SERVICE_MASK services = 0;

    BTIF_TRACE_EVENT1("%s", __FUNCTION__);
    /* TODO: Do we need to handle multiple inquiries at the same time? */

    /* Set inquiry params and call API */
#if (BLE_INCLUDED == TRUE)
    inq_params.mode = BTA_DM_GENERAL_INQUIRY|BTA_BLE_GENERAL_INQUIRY;
#else
    inq_params.mode = BTA_DM_GENERAL_INQUIRY;
#endif
    inq_params.duration = BTIF_DM_DEFAULT_INQ_MAX_DURATION;

    inq_params.max_resps = BTIF_DM_DEFAULT_INQ_MAX_RESULTS;
    inq_params.report_dup = TRUE;

    inq_params.filter_type = BTA_DM_INQ_CLR;
    /* TODO: Filter device by BDA needs to be implemented here */
```

```
    /* Will be enabled to TRUE once inquiry busy level has been received */
    btif_dm_inquiry_in_progress = FALSE;
    /* find nearby devices */
    BTA_DmSearch(&inq_params, services, bte_search_devices_evt);

    return BT_STATUS_SUCCESS;
}
```

在上述代码中，bte_search_devices_evt 函数是核心，此函数的主要实现代码如下所示：

```
static void bte_search_devices_evt(tBTA_DM_SEARCH_EVT event,
  tBTA_DM_SEARCH *p_data) {
    UINT16 param_len = 0;

    if (p_data)
        param_len += sizeof(tBTA_DM_SEARCH);
    /* Allocate buffer to hold the pointers (deep copy). The pointers will point
to the end of the tBTA_DM_SEARCH */
    switch (event)
    {
        case BTA_DM_INQ_RES_EVT:
        {
            if (p_data->inq_res.p_eir)
                param_len += HCI_EXT_INQ_RESPONSE_LEN;
        }
        break;
        ...
    }
    BTIF_TRACE_DEBUG3("%s event=%s param_len=%d", __FUNCTION__,
                      dump_dm_search_event(event), param_len);

    /* if remote name is available in EIR,
    set the flag so that stack doesn't trigger RNR */
    if (event == BTA_DM_INQ_RES_EVT)
        p_data->inq_res.remt_name_not_required =
          check_eir_remote_name(p_data, NULL, NULL);

    btif_transfer_context(btif_dm_search_devices_evt, (UINT16)event,
      (void*)p_data, param_len,
      (param_len>sizeof(tBTA_DM_SEARCH))? search_devices_copy_cb : NULL);
}
```

函数 btif_dm_search_devices_evt 用于在界面展示搜索蓝牙的成果，此函数在文件 external/bluetooth/bluedroid/btif/src/btif_dm.c 中定义，具体实现代码如下所示：

```
static void btif_dm_search_devices_evt(UINT16 event, char *p_param)
{
    tBTA_DM_SEARCH *p_search_data;
    BTIF_TRACE_EVENT2("%s event=%s", __FUNCTION__, dump_dm_search_event(event));
```

```
switch (event)
{
    case BTA_DM_DISC_RES_EVT:
    {
        p_search_data = (tBTA_DM_SEARCH*)p_param;
        /* Remote name update */
        if (strlen((const char*) p_search_data->disc_res.bd_name))
        {
            bt_property_t properties[1];
            bt_bdaddr_t bdaddr;
            bt_status_t status;

            properties[0].type = BT_PROPERTY_BDNAME;
            properties[0].val = p_search_data->disc_res.bd_name;
            properties[0].len = strlen((char*)p_search_data->disc_res.bd_name);
            bdcpy(bdaddr.address, p_search_data->disc_res.bd_addr);

            status = btif_storage_set_remote_device_property(
                        &bdaddr, &properties[0]);
            ASSERTC(status == BT_STATUS_SUCCESS,
              "failed to save remote device property", status);
            HAL_CBACK(bt_hal_cbacks, remote_device_properties_cb,
                        status, &bdaddr, 1, properties);
        }
        /* TODO: Services? */
    }
    break;
    ...
```

函数 BTA_DmSearch()起到了发送消息的作用，在文件 external/bluetooth/bluedroid/bta/dm/bta_dm_api.c 中定义，具体实现代码如下所示：

```
void BTA_DmSearch(tBTA_DM_INQ *p_dm_inq, tBTA_SERVICE_MASK services,
  tBTA_DM_SEARCH_CBACK *p_cback) {
    tBTA_DM_API_SEARCH *p_msg;
    if ((p_msg = (tBTA_DM_API_SEARCH*) GKI_getbuf(sizeof(tBTA_DM_API_SEARCH)))
      != NULL) {
      memset(p_msg, 0, sizeof(tBTA_DM_API_SEARCH));
      p_msg->hdr.event = BTA_DM_API_SEARCH_EVT;
      memcpy(&p_msg->inq_params, p_dm_inq, sizeof(tBTA_DM_INQ));
      p_msg->services = services;
      p_msg->p_cback = p_cback;
      p_msg->rs_res  = BTA_DM_RS_NONE;
      bta_sys_sendmsg(p_msg);
    }
}
```

方法 deviceFoundCallback 最后会来到/packages/apps/Bluetooth/src/com/android/bluetooth/

btservice/RemoteDevices.java 文件中，具体实现代码如下所示：

```
void deviceFoundCallback(byte[] address) {
    // The device properties are already registered - we can send the intent
    // now
    BluetoothDevice device = getDevice(address);
    debugLog("deviceFoundCallback: Remote Address is:" + device);
    DeviceProperties deviceProp = getDeviceProperties(device);
    if (deviceProp == null) {
        errorLog("Device Properties is null for Device:" + device);
        return;
    }

    Intent intent = new Intent(BluetoothDevice.ACTION_FOUND);
    intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);
    intent.putExtra(BluetoothDevice.EXTRA_CLASS,
            new BluetoothClass(Integer.valueOf(deviceProp.mBluetoothClass)));
    intent.putExtra(BluetoothDevice.EXTRA_RSSI, deviceProp.mRssi);
    intent.putExtra(BluetoothDevice.EXTRA_NAME, deviceProp.mName);
    mAdapterService.sendBroadcast(intent, mAdapterService.BLUETOOTH_PERM);
}
```

这样通过界面发送广播，在应用层中会通过 handle 将收到的广播显示出来，这个 handle 可以在文件 BluetoothEventManager.java 的构造函数里找到，具体代码如下所示：

```
addHandler(BluetoothDevice.ACTION_FOUND, new DeviceFoundHandler());
private final BroadcastReceiver mBroadcastReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        BluetoothDevice device =
          intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);

        Handler handler = mHandlerMap.get(action);
        if (handler != null) {
            handler.onReceive(context, intent, device);
        }
    }
};
```

上述 handle 与 DeviceFoundHandler 相对应，具体代码如下所示：

```
private class DeviceFoundHandler implements Handler {
    public void onReceive(Context context, Intent intent,
    BluetoothDevice device) {
        ...
        // TODO Pick up UUID. They should be available for 2.1 devices.
        // Skip for now,
        // there's a bluez problem and we are not getting uuids even for 2.1.
        CachedBluetoothDevice cachedDevice = mDeviceManager.findDevice(device);
```

```
        if (cachedDevice == null) {
            cachedDevice = mDeviceManager.addDevice(
              mLocalAdapter, mProfileManager, device);
            Log.d(TAG, "DeviceFoundHandler created new CachedBluetoothDevice: "
                    + cachedDevice);
            // callback to UI to create Preference for new device
            dispatchDeviceAdded(cachedDevice);
        }
        ...
    }
}
```

在上述 if 语句中，方法 dispatchDeviceAdded()用于向界面分发消息并处理消息，并使用文件/packages/apps/Settings/src/com/android/settings/bluetooth/DeviceListPreferenceFragment.java 实现界面显示功能，对应的代码如下所示：

```
public void onDeviceAdded(CachedBluetoothDevice cachedDevice) {
    if (mDevicePreferenceMap.get(cachedDevice) != null) {
        return;
    }

    // Prevent updates while the list shows one of the state messages
    if (mLocalAdapter.getBluetoothState() != BluetoothAdapter.STATE_ON) return;

    if (mFilter.matches(cachedDevice.getDevice())) {
        createDevicePreference(cachedDevice);
    }
}
```

上述代码的最后一个分支是界面显示需要做的工作，整个过程从 Settings 界面开始，再到 Settings 界面显示搜索到蓝牙结束，对应的代码如下所示：

```
void createDevicePreference(CachedBluetoothDevice cachedDevice) {
    BluetoothDevicePreference preference =
      new BluetoothDevicePreference(getActivity(), cachedDevice);

    initDevicePreference(preference);
    mDeviceListGroup.addPreference(preference);
    mDevicePreferenceMap.put(cachedDevice, preference);
}
```

到目前为止，包括前面的打开流程分析，还仅是针对代码流程做的分析，对于蓝牙协议方面的东西还没有涉及。

14.6.3 传输OPP文件

在 Android 系统中，OPP 文件是蓝牙传输的文件。在分享蓝牙文件的过程中，使用的是蓝牙应用 OPP 目录下的代码。在 Android 设备中，当使用蓝牙发送文件时，发送端先来到文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppLauncherActivity.java 处，这

是一个没有界面，只是提取文件信息的中转站文件，这个文件的核心是两个分支，也就是action.equals(Intent.ACTION_SEND)和 action.equals(Intent.ACTION_SEND_MULTIPLE)，对应的代码如下所示：

```
if (action.equals(Intent.ACTION_SEND)
  || action.equals(Intent.ACTION_SEND_MULTIPLE)) {
    //Check if Bluetooth is available in the beginning instead of at the end
    if (!isBluetoothAllowed()) {
        Intent in = new Intent(this, BluetoothOppBtErrorActivity.class);
        in.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        in.putExtra("title", this.getString(R.string.airplane_error_title));
        in.putExtra("content", this.getString(R.string.airplane error msg));
        startActivity(in);
        finish();
        return;
    }
    if (action.equals(Intent.ACTION_SEND)) {
        ...
        Thread t = new Thread(new Runnable() {
            public void run() {
                BluetoothOppManager
                .getInstance(BluetoothOppLauncherActivity.this)
                .saveSendingFileInfo(type, fileUri.toString(), false);
                //Done getting file info..Launch device picker
                //and finish this activity
                launchDevicePicker();
                finish();
            }
        });
        ...
    } else if (action.equals(Intent.ACTION_SEND_MULTIPLE)) {
        ...
    }
    ...
```

在上述代码中，前面的 isBluetoothAllowed()方法会判断是否处于飞行模式，如果是，则禁止发送 OPP 文件。

launchDevicePicker()中通过条件语句(!BluetoothOppManager.getInstance(this).isEnabled())判断蓝牙是否已经打开，如果已经打开，则进入设备选择界面 DeviceListPreferenceFragment (DevicePickerFragment)来选择设备。

在这个跳转过程中，new Intent(BluetoothDevicePicker.ACTION_LAUNCH)中字符串的完整定义是：

```
public static final String ACTION_LAUNCH =
  "android.bluetooth.devicepicker.action.LAUNCH";
```

对应的文件是 frameworks/base/core/java/android/bluetooth/BluetoothDevicePicker.java，在setting 应用的 manifest.xml 文件中会发现，对应代码如下所示：

```
<activity android:name=".bluetooth.DevicePickerActivity"
  android:theme="@android:style/Theme.Holo.DialogWhenLarge"
  android:label="@string/device_picker"
  android:clearTaskOnLaunch="true">
    <intent-filter>
        <action android:name="android.bluetooth.devicepicker.action.LAUNCH" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

上述代码指向了 DevicePickerActivity，其源码路径是：

```
packages/apps/Settings/src/com/android/settings/bluetooth/DevicePickerActivity.java
```

类 DevicePickerActivity 的实现代码很简单，只有一个 onCreate，并且只在里面加载了一个布局文件 bluetooth_device_picker.xml，对应的代码如下所示：

```
<fragment
  android:id="@+id/bluetooth_device_picker_fragment"
  android:name="com.android.settings.bluetooth.DevicePickerFragment"
  android:layout_width="match_parent"
  android:layout_height="0dip"
  android:layout_weight="1" />
```

上述布局文件设置了下一步的位置是 DevicePickerFragment，此时可以看到配对后的蓝牙列表，列表中的 sendDevicePickedIntent 会又发送一个广播，对应代码如下所示：

```
void onDevicePreferenceClick(BluetoothDevicePreference btPreference) {
    mLocalAdapter.stopScanning();
    LocalBluetoothPreferences.persistSelectedDeviceInPicker(
      getActivity(), mSelectedDevice.getAddress());
    if ((btPreference.getCachedDevice().getBondState()
      == BluetoothDevice.BOND_BONDED) || !mNeedAuth) {
        sendDevicePickedIntent(mSelectedDevice);
        finish();
    } else {
        super.onDevicePreferenceClick(btPreference);
    }
}
public static final String ACTION_LAUNCH =
  "android.bluetooth.devicepicker.action.LAUNCH";
private void sendDevicePickedIntent(BluetoothDevice device) {
    Intent intent = new Intent(BluetoothDevicePicker.ACTION_DEVICE_SELECTED);
    intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);
    if (mLaunchPackage!=null && mLaunchClass!=null) {
        intent.setClassName(mLaunchPackage, mLaunchClass);
    }
    getActivity().sendBroadcast(intent);
}
```

通过 BluetoothDevicePicker.ACTION_DEVICE_SELECTED 查找，会在文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppReceiver.java 中找到对上述发送广播的处理，对应的代码如下所示：

```
else if (action.equals(BluetoothDevicePicker.ACTION_DEVICE_SELECTED)) {
    BluetoothOppManager mOppManager = BluetoothOppManager.getInstance(context);
    BluetoothDevice remoteDevice =
      intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
    // Insert transfer session record to database
    mOppManager.startTransfer(remoteDevice);
    // Display toast message
    String deviceName = mOppManager.getDeviceName(remoteDevice);
    ...
}
```

在上述代码中，调用的方法 mOppManager.startTransfer(remoteDevice)在文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppManager.java 中实现，功能是开启线程，执行发送动作，通过 run 方法分单个或多个文件进行发送，其中单个文件的发送代码如下所示：

```
public void startTransfer(BluetoothDevice device) {
    if (V)
      Log.v(TAG, "Active InsertShareThread number is : " + mInsertShareThreadNum);
    InsertShareInfoThread insertThread;
    synchronized(BluetoothOppManager.this) {
        if (mInsertShareThreadNum > ALLOWED_INSERT_SHARE_THREAD_NUMBER) {
            ...
            return;
        }
        insertThread = new InsertShareInfoThread(
          device, mMultipleFlag, mMimeTypeOfSendingFile,
          mUriOfSendingFile, mMimeTypeOfSendingFiles,
          mUrisOfSendingFiles, mIsHandoverInitiated);
        if (mMultipleFlag) {
            mfileNumInBatch = mUrisOfSendingFiles.size();
        }
    }
    insertThread.start();
}
public void run() {
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    ...
    if (mIsMultiple) {
        insertMultipleShare();
    } else {
        insertSingleShare();
    }
    ...
}
```

以 insertSingleShare()的实现过程为例，调用了方法 mContext.getContentResolver().insert，此方法在文件 bluetooth\src\com\android\bluetooth\opp\BluetoothOppProvider.java 中定义，具体代码如下所示：

```
public Uri insert(Uri uri, ContentValues values) {
    if (rowID != -1) {
        context.startService(new Intent(context, BluetoothOppService.class));
        ret = Uri.parse(BluetoothShare.CONTENT_URI + "/" + rowID);
        context.getContentResolver().notifyChange(uri, null);
    } else {
        if (D) Log.d(TAG, "couldn't insert into btopp database");
    }
    ...
```

由上述代码可知，又调用了 BluetoothOppService 服务，实现文件是文件 packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppService.java。

在文件 BluetoothOppService.java 的方法 onStartCommand 中会看到 updateFromProvider()方法，通过此方法开启一个 UpdateThread 线程，并通过 run 方法执行到 BluetoothOppTransfer.java 的对象，对应的代码如下所示：

```
private void insertShare(Cursor cursor, int arrayPos) {
    ...
    /*
    * Add info into a batch. The logic is
    * 1) Only add valid and readyToStart info
    * 2) If there is no batch, create a batch and insert this transfer into batch,
    * then run the batch
    * 3) If there is existing batch and timestamp match, insert transfer into batch
    * 4) If there is existing batch and timestamp does not match,
    * create a new batch and put in queue
    */
    if (info.isReadyToStart()) {
        ...
        if (mBatchs.size() == 0) {
            ...
            mBatchs.add(newBatch);
            if (info.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
                mTransfer=new BluetoothOppTransfer(this,mPowerManager,newBatch);
            } else if (info.mDirection == BluetoothShare.DIRECTION_INBOUND) {
                mServerTransfer = new BluetoothOppTransfer(
                  this, mPowerManager, newBatch, mServerSession);
            }
            if (info.mDirection == BluetoothShare.DIRECTION_OUTBOUND
              && mTransfer != null) {
                mTransfer.start();
            } else if (info.mDirection == BluetoothShare.DIRECTION_INBOUND
              && mServerTransfer != null) {
                mServerTransfer.start();
```

```
            }
        } else {
            ...
        }
    }
    ...
```

方法 start()并不是线程方法，其实现文件是：

```
packages/apps/Bluetooth/src/com/android/bluetooth/opp/BluetoothOppTransfer.java
```

对应的代码如下所示：

```
public void start() {
    ....这里省略未贴的代码是检查蓝牙是否打开，提高了安全

    if (mHandlerThread == null) {
        ...
        if (mBatch.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
            /* for outbound transfer, we do connect first */
            startConnectSession();
        } else if (mBatch.mDirection == BluetoothShare.DIRECTION_INBOUND) {
            /*
            * for inbound transfer, it's already connected, so we start
            * OBEX session directly
            */
            startObexSession();
        }
    }
}
```

上述代码用于发送文件和接收文件，如果分享给别人，则是 OUTBOUND，会先执行 startConnectSession()。如果收文件直接 startObexSession，startConnectSession()函数最后还是要执行到 startObexSession()处，对应的代码如下所示：

```
public static final int DIRECTION_OUTBOUND = 0;
// This transfer is inbound, e.g. receive file from other device.
public static final int DIRECTION_INBOUND = 1;
```

发送文件功能在 BluetoothOppObexClientSession.java 中开启，对应的代码如下所示：

```
private void startObexSession() {
    if (mBatch.mDirection == BluetoothShare.DIRECTION_OUTBOUND) {
        if (V) Log.v(TAG,
          "Create Client session with transport " + mTransport.toString());
        mSession = new BluetoothOppObexClientSession(mContext, mTransport);
    } else if (mBatch.mDirection == BluetoothShare.DIRECTION_INBOUND) {
        if (mSession == null) {
            markBatchFailed();
            mBatch.mStatus = Constants.BATCH_STATUS_FAILED;
            return;
```

```
        }
        if (V) Log.v(TAG, "Transfer has Server session" + mSession.toString());
    }
    mSession.start(mSessionHandler);
    processCurrentShare();
}
```

真正的发送文件功能在如下文件中实现：

```
packages/apps/Bluetooth/src/com/android/bluetooth/opp/
BluetoothOppObexClientSession.java
```

对应的代码如下所示：

```
private void doSend() {

    int status = BluetoothShare.STATUS_SUCCESS;
    ........省略关于 status 值的判断
    if (status == BluetoothShare.STATUS_SUCCESS) {
        /* do real send */ // sendFile
        if (mFileInfo.mFileName != null) {
            status = sendFile(mFileInfo);
        } else {
            /* this is invalid request */
            status = mFileInfo.mStatus;
        }
        waitingForShare = true;
    } else {
        Constants.updateShareStatus(mContext1, mInfo.mId, status);
    }

    if (status == BluetoothShare.STATUS_SUCCESS) {
        Message msg = Message.obtain(mCallback);
        msg.what = BluetoothOppObexSession.MSG_SHARE_COMPLETE;
        msg.obj = mInfo;
        msg.sendToTarget();
    } else {
        Message msg = Message.obtain(mCallback);
        msg.what = BluetoothOppObexSession.MSG_SESSION_ERROR;
        mInfo.mStatus = status;
        msg.obj = mInfo;
        msg.sendToTarget();
    }
}
```

当执行完 sendFile 后，会把分享成功或失败的消息传回去，在 sendFile 中会执行打包的过程，对于各个字段的具体说明，需要分析文件 frameworks/base/obex/javax/obex/HeaderSet.java，到此为止，蓝牙发送文件的基本流程讲解完毕。

在蓝牙接收文件的过程中，会收到 MSG_INCOMING_BTOPP_CONNECTION 消息，这是因为在蓝牙打开时(即蓝牙状态是 BluetoothAdapter.STATE_ON 时)会执行 startSocketListener()

方法，在此函数中开启了监听程序，对应的代码如下所示：

```
private void createServerSession(ObexTransport transport) {
    mServerSession = new BluetoothOppObexServerSession(this, transport);
    mServerSession.preStart();
}
```

到此为止，对于蓝牙接收文件部分的流程不会再详细分析，因为与发送流程的原理完全一样。要想更好地理解蓝牙 OPP 文件的传输过程，需要了解 OBEX 基础协议方面的知识，网上有很多相关的论文资料。

Android

第 15 章
网络系统详解

随着计算机技术的不断发展，互联网已经成为人们生活中必不可少的一部分。如今网上冲浪、QQ 交友、微信聊天、天猫购物等应用已经到处可见，互联网在不知觉间改变了人们的生活。Android 作为一款移动智能设备系统，很自然地具备互联网功能。在本章的内容中，我们将详细讲解 Android 4.3 系统中与网络应用相关的源码，为读者深入理解 Android 网络系统的架构打下基础。

15.1 使用WebKit浏览网页

在 Android 系统中，WebKit 是一个网络引擎库，其主要功能是实现 Web 浏览器的引擎，达到浏览网页数据的目的。在 Android 系统中，浏览 Web 浏览器和一个可嵌入的 Web 视图的功能是通过 WebKit 实现的，这是一个第三方开发的浏览器引擎。WebKit 的源码被保存在/external/webkit/目录下，其目录结构如下所示：

```
external/webkit/
├── Examples                    //WebKit 的例子
├── LayoutTests                 //布局测试
├── PerformanceTests            //表现测试
├── Source                      //WebKit 的源代码
├── Tools                       //工具
├── WebKitLibraries             //WebKit 用到的库
├── Android.mk                  //Makefile
├── bison_check.mk
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL         //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION       //版本信息
```

为了从更加深层次中了解 WebKit 浏览器编程的基本知识，本书将首先从 Android 底层开始分析 WebKit 系统的机理和用法，依次从下到上分析 WebKit 浏览器编程的基本知识。在 Android 系统中，WebKit 模块分成 Java 和 WebKit 库两个部分，具体说明如下所示。

- Java 层：负责与 Android 应用程序进行通信。
- WebKit 类库：因为是由 C/C++实现的，所以也被称为 C 层库，WebKit 类库部分负责实际的网页排版处理。

Java 层和 WebKit 类库之间通过 JNI 和 Bridge 实现相互调用，如图 15-1 所示。

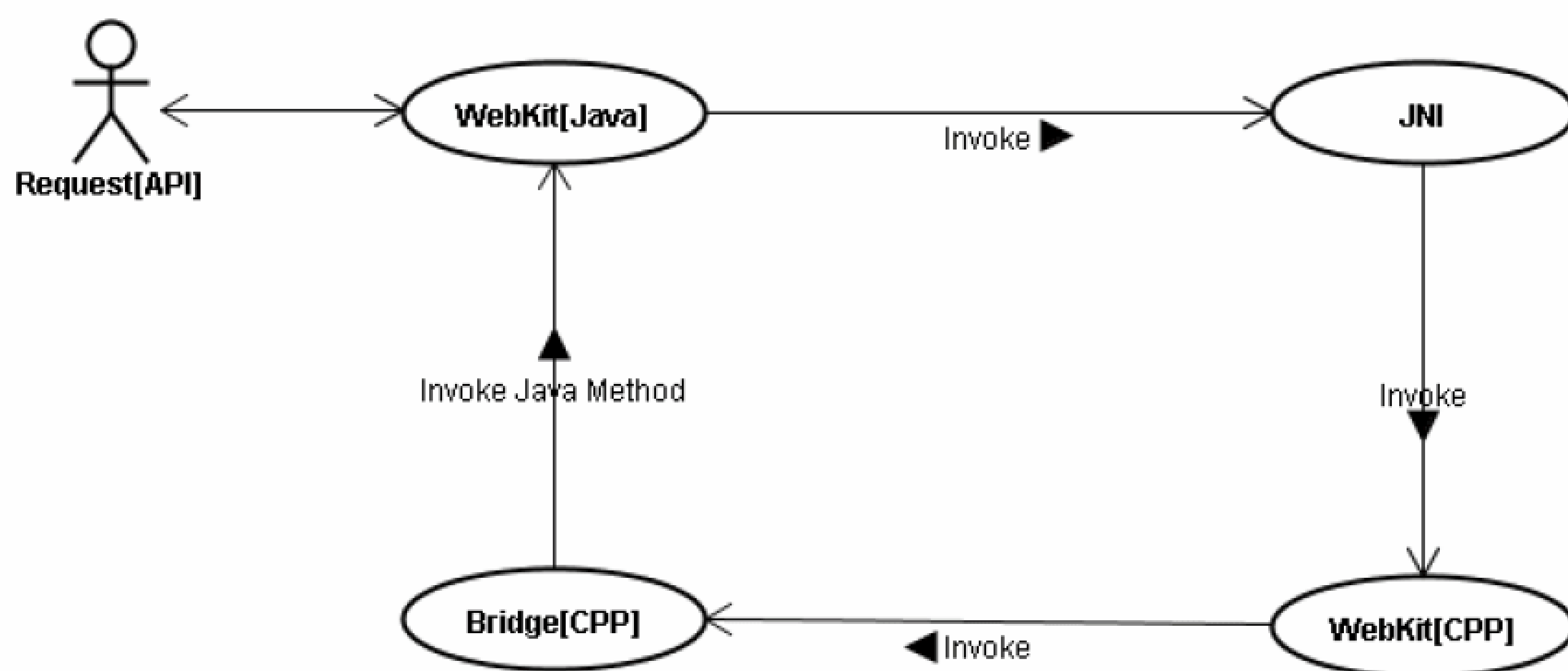


图 15-1　WebKit系统的框架结构

在本节的内容中，将详细分析 Android 4.3 系统中 WebKit 网络引擎库的基本源码。

15.1.1　WebKit的Java层框架

在 Android 系统中，WebKit 模块中 Java 层的根目录是：

```
\frameworks\base\core\java\android\webkit\
```

上述目录是基于 Android 4.3 的，其目录结构如表 15-1 所示。

表 15-1　WebKit的目录结构

目录中相关的文件	文件的具体用途
BrowserFrame.java	BrowserFrame 对象是对 WebCore 库中的 Frame 对象的 Java 层封装，用于创建 WebCore 中定义的 Frame，以及为该 Frame 对象提供 Java 层回调方法
ByteArrayBuilder.java	ByteArrayBuilder 辅助对象，用于 byte 块链表的处理
CachLoader.java	URL Cache 载入器对象，该对象实现 StreadLoader 抽象基类，用于通过 CacheResult 对象载入内容数据
CacheManager.java	Cache 管理对象，负责 Java 层 Cache 对象管理
CacheSyncManager.java	Cache 同步管理对象，负责同步 RAM 和 Flash 之间的浏览器 Cache 数据。实际的物理数据操作在 WebSyncManager 对象中完成
CallbackProxy.java	该对象是用于处理 WebCore 与 UI 线程消息的代理类。当有 Web 事件产生时，WebCore 线程会调用该回调代理类，代理类会通过消息的方式通知 UI 线程，并且调用设置的客户对象的回调函数
CellList.java	CellList 定义图片集合中的 Cell，管理 Cell 图片的绘制、状态改变以及索引
CookieManager.java	根据 RFC2109 规范来管理 Cookies
CookieSyncManager.java	Cookies 同步管理对象，该对象负责同步 RAM 和 Flash 之间的 Cookies 数据。实际的物理数据操作在基类 WebSyncManager 中完成
DataLoader.java	数据载入器对象，用于载入网页数据
DateSorter.java	尚未使用
DownloadListener.java	下载侦听器接口
DownloadManagerCore.java	下载管理器对象，管理下载列表。该对象运行在 WebKit 的线程中，通过 CallbackProxy 对象与 UI 线程交互
FileLoader.java	文件载入器，将文件数据载入到 Frame 中
FrameLoader.java	Frame 载入器，用于载入网页 Frame 数据
HttpAuthHandler.java	HTTP 认证处理对象，该对象会作为参数传递给 BrowserCallback.displayHttpAuthDialog 方法，与用户交互
HttpDataTime.java	该对象是处理 HTTP 日期的辅助对象
JSConfirmResult.java	JS 确认请求对象
JSPromptResult.java	JS 结果提示对象，用于向用户提示 JavaScript 运行结果
JSResult.java	JS 结果对象，用于实现用户交互
JWebCoreJavaBridge.java	用 Java 与 WebCore 库中 Timer 和 Cookies 对象交互的桥接代码

续表

目录中相关的文件	文件的具体用途
LoadListener.java	载入器侦听器，用于处理载入器侦听消息
Network.java	该对象封装网络连接逻辑，为调用者提供更为高级的网络连接接口
PanZoom.java	用于处理图片缩放、移动等操作
PanZoomCellList.java	用于保存移动、缩放图片的 Cell
SslErrorHandler.java	用于处理 SSL 错误消息
StreamLoader.java	StreamLoader 抽象类是所有内容载入器对象的基类。该类是通过消息方式控制的状态机，用于将数据载入到 Frame 中
TextDialog.java	用于处理 HTML 中的文本区域叠加情况，可以使用通过标准的文本编辑而定义的特殊 EditText 控件
URLUtil.java	URL 处理功能函数，用于编码、解码 URL 字符串，以及提供附加的 URL 类型分析功能
WebBackForwardList.java	该对象包含 WebView 对象中显示的历史数据
WebBackForwardListClient.java	浏览历史处理的客户接口类，所有需要接收浏览历史改变的类都需要实现该接口
WebChromeClient.java	Chrome 客户基类，Chrome 客户对象在浏览器文档标题、进度条、图标改变时会得到通知
WebHistoryItem.java	该对象用于保存一条网页历史数据
WebIconDataBase.java	图表数据库管理对象，所有的 WebView 均请求相同的图标数据库对象
WebSettings.java	WebView 的管理设置数据，该对象数据是通过 JNI 接口从底层获取的
WebSyncManager.java	数据同步对象，用于 RAM 数据和 Flash 数据的同步操作
WebView.java	Web 视图对象，用于基本的网页数据载入、显示等 UI 操作
WebViewClient.java	Web 视图客户对象，在 Web 视图中有事件产生时，该对象可以获得通知
WebViewCore.java	该对象对 WebCore 库进行了封装，将 UI 线程中的数据请求发送给 WebCore 处理，并且以 CallbackProxy 的方式，通过消息通知 UI 线程数据处理的结果
WebViewDatabase.java	该对象使用 SQLiteDatabase 为 WebCore 模块提供数据存取操作

在接下来的内容中，将对 WebKit 模块的 Java 层的具体知识进行详细介绍。

1. 主要类

WebKit 模块的 Java 层一共由 41 个文件组成，其中主要类的具体说明如下所示。

(1) WebView

类 WebView 是 WebKit 模块 Java 层的视图类，所有需要使用 Web 浏览功能的 Android 应用程序都要创建该视图对象以显示和处理请求的网络资源。目前，WebKit 模块支持 HTTP、HTTPS、FTP 以及 JavaScript 请求。WebView 作为应用程序的 UI 接口，为用户提供了一系列的网页浏览、用户交互接口，客户程序通过这些接口访问 WebKit 核心代码。

在文件 WebView.java 中，类 WebView 的主要实现代码如下所示：

```
public class WebView extends AbsoluteLayout
  implements ViewTreeObserver.OnGlobalFocusChangeListener,
  ViewGroup.OnHierarchyChangeListener, ViewDebug.HierarchyHandler {

    private static final String LOGTAG = "webview_proxy";

    // Throwing an exception for incorrect thread usage if the
    // build target is JB MR2 or newer. Defaults to false, and is
    // set in the WebView constructor.
    private static Boolean sEnforceThreadChecking = false;

    /**
    *  Transportation object for returning WebView across thread boundaries.
    */
    public class WebViewTransport {
        private WebView mWebview;
        public synchronized void setWebView(WebView webview) {
            mWebview = webview;
        }
        public synchronized WebView getWebView() {
            return mWebview;
        }
    }
    public static final String SCHEME_TEL = "tel:";
    public static final String SCHEME_MAILTO = "mailto:";
    public static final String SCHEME_GEO = "geo:0,0?q=";
    ...
```

注意： 类 WebView 是一个非常重要的类，能够实现与网络有关的很多功能。为了节省本书的篇幅，后面各个 Java 类的实现代码将不再一一列出。

(2) WebViewDatabase

类 WebViewDatabase 是 WebKit 模块中针对 SQLiteDatabase 对象的封装，用于存储和获取运行时浏览器保存的缓冲数据、历史访问数据、浏览器配置数据等。该对象是一个单实例对象，通过 getInstance 方法获取 WebViewDatabase 的实例。WebViewDatabase 是 WebKit 模块中的内部对象，仅供 WebKit 框架内部使用。

(3) WebViewCore

类 WebViewCore 是 Java 层与 C 层 WebKit 核心库的交互类，客户程序调用 WebView 的网页浏览相关操作会转发给 BrowserFrame 对象。当 WebKit 核心库完成实际的数据分析和处理后会回调 WebViweCore 中定义的一系列 JNI 接口，这些接口会通过 CallbackProxy 将相关事件通知相应的 UI 对象。

(4) CallbackProxy

类 CallbackProxy 是一个代理类，用于实现 UI 线程和 WebCore 线程之间的交互。

类 CallbackProxy 定义了一系列与用户相关的通知方法，当 WebCore 完成相应的数据处理

后，会调用 CallbackProxy 类中对应的方法，这些方法通过消息方式间接调用相应处理对象的处理方法。

(5) BrowserFrame

类 BrowserFrame 负责 URL 资源的载入、访问历史的维护、数据缓存等操作，该类会通过 JNI 接口直接与 WebKit C 层库交互。

(6) JWebCoreJavaBridge

类 JWebCoreJavaBridge 为 Java 层 WebKit 代码提供与 C 层 WebKit 核心部分的 Timer 和 Cookies 操作相关的方法。

(7) DownloadManagerCore

类 DownloadManagerCore 是一个下载管理核心类，主要负责管理网络资源的下载，所有的 Web 下载操作均由该类统一管理。该类实例运行在 WebKit 线程中，与 UI 线程的交互是通过调用 CallbackProxy 对象中相应的方法完成的。

(8) WebSettings

类 WebSettings 描述了 Web 浏览器访问相关的用户配置信息。

(9) DownloadListener

类 DownloadListener 负责下载侦听接口，如果客户代码实现该接口，则在下载开始、失败、挂起、完成等情况下，DownloadManagerCore 对象会调用客户代码中实现的 DwonloadListener 方法。

(10) WebBackForwardList

类 WebBackForwarList 负责维护用户访问的历史记录，该类为客户程序提供操作访问浏览器历史数据的相关方法。

(11) WebViewClient

在类 WebViewClient 中定义了一系列事件方法，如果 Android 应用程序设置了 WebViewClient 派生对象，则在页面载入、资源载入、页面访问错误等情况发生时，该派生对象的相应方法会被调用。

(12) WebBackForwardListClient

类 WebBackForwardListClient 定义了访问历史操作时可能产生的事件接口，当用户实现了该接口后，则在操作访问历史时(访问历史移除、访问历史清空等)，用户会得到通知。

(13) WebChromeClient

类 WebChromeClient 定义了与浏览窗口修饰相关的事件。例如接收到 Title、接收到 Icon、进度变化时，WebChromeClient 的相应方法会被调用。

2. 数据载入器的设计理念

在 WebKit 系统的 Java 部分框架中，使用数据载入器来加载相应类型的数据，目前有 CacheLoader、DataLoader 以及 FileLoader 三类载入器，它们分别用于处理缓存数据、内存数据，以及文件数据的载入操作。Java 层(WebKit 模块)所有的载入器都从 StreamLoader 继承(其父类为 Handler)，由于 StreamLoader 类的基类为 Handler 类，因此在构造载入器时，会开启一个事件处理线程，该线程负责实际的数据载入操作，而请求线程通过消息的方式驱动数据的载入。

图 15-2 描述了数据载入器相关类的类图结构。

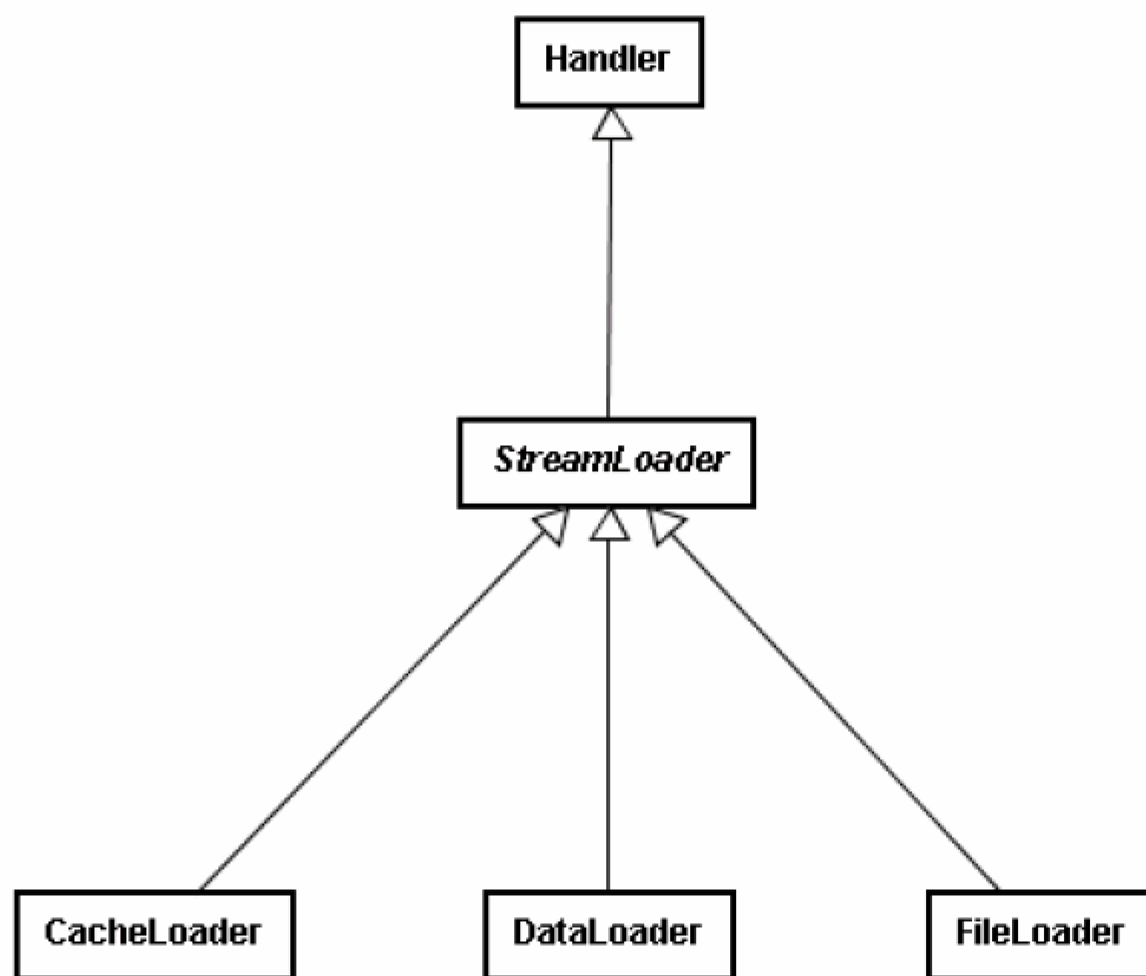


图 15-2　数据载入器的类图结构

在类 StreamLoader 中，定义了如下 4 个不同的消息。

- MSG_STATUS：表示发送状态消息。
- MSG_HEADERS：表示发送消息头消息。
- MSG_DATA：表示发送数据消息。
- MSG_END：表示数据发送完毕消息。

在类 StreamLoader 中提供了两个抽象保护方法以及一个共有方法，其中保护方法 setupStreamAndSendStatus 用于构造与通信协议相关的数据流，以及向 LoadListener 发送状态。

方法 buildHeaders 负责向子类提供构造特定协议消息头的功能。所有载入器只有一个共有方法(load)，因此当需要载入数据时，只需调用该方法即可。与数据载入流程相关的类还有 LoaderListener 和 BrowserFrame，当发生数据载入事件时，WebKit 的 C 库会更新载入进度，并且会通知 BrowserFrame，BroserFrame 接收到进度条变更事件后，会通过 CallbackProxy 对象通知 View 类进度条数据变更。

15.1.2　C/C++层框架

因为 C 层框架属于 Android 体系底层的知识，而我们本书主要讲解了 Android 在 Java 层开发网络应用的知识，所以在此简要介绍 WebKit 系统 C 层框架的基本知识，只简单分析 C 层框架中各个类之间的关系。读者了解了这些类之间的关系和原理后，当在 Java 层开发应用时，即可达到“游刃有余”。

1. Java层对应的C/C++类库

在介绍 Java 层时，每一个 Java 类在下面的 C/C++层都会有一个对应的类库，各个 Java 类与 C/C++类库的对应关系的具体说明如表 15-2 所示。

表 15-2　Java层中的类和C/C++类库的对应关系

类	功能描述
ChromeClientAndroid	该类主要处理 WebCore 中与 Frame 装饰相关的操作。例如设置状态栏、滚动条、JavaScript 脚本提示框等。 当浏览器中有相关事件产生时，ChromeClientAndroid 类的相应方法会被调用，该类会将相关的 UI 事件通过 Bridge 传递给 Java 层，由 Java 层负责绘制以及用户交互方面的处理
EditorClientAndroid	该类负责处理页面中与文本相关的处理，比如文本输入、取消、输入法数据处理、文本粘贴、文本编辑等操作。不过目前该类只对按键相关的时间进行了处理，其他操作均未支持
ContextMenuClient	该类提供页面相关的功能菜单，比如图片拷贝、朗读、查找等功能。但是，目前项目中未实现具体的功能
DragClient	该类定义了与页面拖拽相关的处理，但是目前该类没有实现具体的功能
FrameLoaderClientAndroid	该类提供与 Frame 加载相关的操作，当用户请求加载一个页面时，WebCore 分析完网页数据后，会通过该类调用 Java 层的回调方法，通知 UI 相关的组件处理
InspectorClientAndroid	该类提供与窗口相关的操作，比如窗口显示、关闭窗口、附加窗口等。不过目前该类的各个方法均为空实现
Page	该类提供与页面相关的操作，比如网页页面的前进、后退等操作
FrameAndroid	该类为 Android 提供 Frame 管理
FrameBridge	该类对 Frame 相关的 Java 层方法进行了封装，当有 Frame 事件产生时，WebCore 通过 FrameBridge 回调 Java 的回调函数，完成用户交互过程
AssetManager	该类为浏览器提供本地资源访问功能
RenderSkinAndroid	该类与控件绘制相关，所有的须绘制控件都需要从该类派生，目前 WebKit 模块中有 Button、Combo、Radio 三类控件

在接下来的内容中，将详细讲解 WebKit 中 C/C++层库的基本知识。

(1)　BrowserFrame

与 Java 类 BrowserFrame 相对应的 C++类为 FrameBridge，该类为 Dalvik 虚拟机回调 BrowserFrame 类中定义的本地方法进行了封装。与 BrowserFrame 中回调函数(Java 层)相对应的 C 层结构定义代码如下所示：

```
struct FrameBridge::JavaBrowserFrame
{
    JavaVM *mJVM;
    jobject mObj;
    jmethodID mStartLoadingResource;
    jmethodID mLoadStarted;
    jmethodID mUpdateHistoryForCommit;
    jmethodID mUpdateCurrentHistoryData;
    jmethodID mReportError;
```

```
    jmethodID setTitle;
    jmethodID mWindowObjectCleared;
    jmethodID mDidReceiveIcon;
    jmethodID mUpdateVisiteHistory;
    jmethodID mHandleUrl;
    jmethodID mCreateWindow;
    jmethodID mCloseWindow;
    jmethodID mDecidePolicyForFormResubmission;
};
```

在上述代码结构中，mJavaFrame 作为 FrameBridge(C 层)的一个成员变量，在 FrameBridge 构造函数中，用类 BrowserFrame(Java 层)的回调方法的偏移量初始化 JavaBrowserFrame 结构的各个域。初始工作完成后，当 WebCore(C 层)在剖析网页数据时，与 Frame 相关的资源会发生改变(比如 Web 页面的主题变化)，此时会通过 mJavaFrame 结构调用指定 BrowserFrame 对象的相应方法，并通知 Java 层进行处理。

注意：　为了节省本书的篇幅，后面各个类库的实现代码将不再一一列出。

(2)　JWebCoreJavaBridge

与该对象相对应的 C 层对象为 JavaBridge，JavaBridge 对象继承了 TimerClient 和 CookieClient 类，负责 WebCore 中的定时器和 Cookie 管理。与 Java 层 JWebCoreJavaBridge 类中方法偏移量相关的是 JavaBridege 中的几个成员变量，在构造 JavaBridge 对象时，会初始化这些成员变量，之后有 Timer 或者 Cookies 事件产生，WebCore 会通过这些 ID 值回调对应 JWebCoreJavaBridge 的相应方法。

(3)　LoadListener

与该对象相关的 C 层结构是 struct resourceloader_t，该结构保存了 LoadListener 对象 ID、CancelMethod ID 以及 DownloadFiledMethod ID 的值。当有 Cancel 或者 Download 事件产生时，WebCore 会回调 LoadListener 类中的 CancelMethod 或者 DownloadFileMethod。

(4)　WebViewCore

与 WebViewCore 相关的 C 类是 WebCoreViewImpl，WebViewCoreImpl 类有个 JavaGlue 对象作为成员变量，在构建 WebCoreViewImpl 对象时，用 WebViewCore(Java 层)中的方法 ID 值初始化该成员变量，并且会将构建的 WebCoreViewImpl 对象指针复制给 WebViewCore(Java 层)的 mNativeClass，这样就能将 WebViewCore(Java 层)与 WebViewCoreImple(C 层)关联起来。

(5)　WebSettings

与 WebSettings 相关的 C 层结构是 struct FieldIds，该结构保存了 WebSettings 类中定义的属性 ID 以及方法 ID，在 WebCore 初始化时(WebViewCore 的静态方法中使用 System.loadLibrary 载入)会设置这些方法和属性的 ID 值。

(6)　WebView

与 WebView 相关的 C 层类是 WebViewNative，在该类中的 mJavaGlue 中保存着 WebView 中定义的属性和方法 ID，在 WebViewNative 构造方法中初始化，并且将构造的 WebViewNative 对象的指针赋值给 WebView 类的 mNativeClass 变量，这样 WebView 与 WebViewNative 对象就建立了关系。

2. 其他的类

接下来总结与 Java 层相关的 C 层类，具体信息如下所示。

- ChromeClientAndroid：该类主要处理 WebCore 中与 Frame 装饰相关的操作。例如设置状态栏、滚动条、JavaScript 脚本提示框等。当浏览器中有相关事件产生时，ChromeClientAndroid 类的相应方法会被调用，该类会将相关的 UI 事件通过 Bridge 传递给 Java 层，由 Java 层负责绘制以及用户交互方面的处理。
- EditorClientAndroid：该类负责处理页面中文本相关的处理，比如文本输入、取消、输入法数据处理、文本粘贴、文本编辑等操作。不过目前该类只对按键相关的事件进行了处理，其他操作均未支持。
- ContextMenuClient：该类提供页面相关的功能菜单，比如图片拷贝、朗读、查找等功能。但是，目前项目中未实现具体功能。
- DragClient：该类定义了与页面拖拽相关的处理，但是目前该类没有实现具体功能。
- FrameLoaderClientAndroid：该类提供与 Frame 加载相关的操作，当用户请求加载一个页面时，WebCore 分析完网页数据后，会通过该类调用 Java 层的回调方法，通知 UI 相关的组件处理。
- InspectorClientAndroid：该类提供与窗口相关的操作，比如窗口显示、关闭窗口、附加窗口等。不过目前该类的各个方法均为空实现。
- Page：该类提供与页面相关的操作，比如网页页面的前进、后退等操作。
- FrameAndroid：该类为 Android 提供 Frame 管理。
- FrameBridge：该类对 Frame 相关的 Java 层方法进行了封装，当有 Frame 事件产生时，WebCore 通过 FrameBridge 回调 Java 的回调函数，完成用户交互过程。
- AssetManager：该类为浏览器提供本地资源访问功能。
- RenderSkinAndroid：该类与控件绘制相关，所有的须绘制控件都需要从该类派生，目前 WebKit 模块中有 Button、Combo、Radio 三类控件。

上述类会在 Java 层请求创建 Web Frame 的时候被建立。

15.1.3 分析WebKit的操作过程

经过对本章前面内容的学习，我们已经基本了解了 WebKit 系统中各层主要类的功能。在接下来的内容中，将简单介绍与 WebKit 相关的基本操作知识，为读者步入本书后面知识的学习打下基础。

1. WebKit初始化

在 Android SDK 中，提供了 WebView 类，使用此类可以提供自定义的浏览显示功能。如果客户需要加入浏览器的支持，可将该类的实例或者派生类的实例作为视图，调用 Activity 类的 setContentView 显示给用户。当客户代码中第一次生成 WebView 对象时，会初始化 WebKit 库(包括 Java 层和 C 层两个部分)，之后用户可以操作 WebView 对象，完成网络或者本地资源的访问。

WebView 对象的生成主要涉及 3 个类：CallbackProxy、WebViewCore 及 WebViewDatabase。

其中，CallbackProxy 对象为 WebKit 模块中的 UI 线程和 WebKit 类库提供交互功能，WebViewCore 是 WebKit 的核心层，负责与 C 层交互以及 WebKit 模块 C 层类库初始化，而 WebViewDatabase 为 WebKit 模块运行时缓存、数据存储提供支持。

初始化的过程就是使用 WebView 创建 CallbackProxy 对象和 WebViewCore 对象的过程。WebKit 模块的初始化流程如下所示。

① 调用 System.loadLibrary 载入 WebCore 相关类库(C 层)。

② 如果是第一次初始化 WebViewCore 对象，则创建 WebCoreTherad 线程。

③ 创建 EventHub 对象，处理 WebViewCore 事件。

④ 获取 WebIconDatabase 对象实例。

⑤ 向 WebCoreThread 发送初始化消息。

根据上述流程，假如我们要获取 WebViewDatabase 实例，则可以按照下面的步骤来实现。

(1) 调用 System.loadLibrary 方法载入 WebCore 相关类库，该过程由 Dalvik 虚拟机完成，它会从动态链接库目录中寻找 libWebCore.so 类库，载入到内存中，并且调用 WebKit 初始化模块的 JNI_OnLoad 方法。WebKit 模块的 JNI_OnLoad 方法中完成了如下所示的初始化操作。

- 初始化 framebridge[register_android_webcore_framebridge]：初始化 gFrameAndroidField 静态变量，以及注册 BrowserFrame 类中的本地方法表。
- 初始化 javabridge[register_android_webcore_javabridge]：初始化 gJavaBridge.mObject 对象，以及注册 JWebCoreJavaBridge 类中的本地方法。
- 初始化资源 loader[register_android_webcore_resource_loader]：初始化 gResourceLoader 静态变量，以及注册 LoadListener 类的本地方法。
- 初始化 webviewcore[register_android_webkit_webviewcore]：初始化 gWebCoreView-ImplField 静态变量，以及注册 WebViewCore 类的本地方法。
- 初始化 webhistory[register_android_webkit_webhistory]：初始化 gWebHistoryItem 结构，以及注册 WebBackForwardList 和 WebHistoryItem 类的本地方法。
- 初始化 webicondatabase[register_android_webkit_webicondatabase]：注册 WebIconData-base 类的本地方法。
- 初始化 websettings[register_android_webkit_websettings]：初始化 gFieldIds 静态变量，以及注册 WebSettings 类的本地方法。
- 初始化 webview[register_android_webkit_webview]：初始化 gWebViewNativeField 静态变量，以及注册 WebView 类的本地方法。

(2) 实现 WebCoreThread 初始化，该初始化只在第一次创建 WebViewCore 对象时完成，当用户代码第一次生成 WebView 对象时，会在初始化 WebViewCore 类时创建 WebCoreThread 线程，该线程负责处理 WebCore 初始化事件。此时 WebViewCore 构造函数会被阻塞，直到一个 WebView 初始化请求完毕时，会在 WebCoreThread 线程中唤醒。

(3) 创建 EventStub 对象，该对象处理 WebView 类的事件，当 WebCore 初始化完成后，会向 WebView 对象发送事件，WebView 类的 EventStub 对象处理该事件，并且完成后续的初始化工作。

(4) 获取 WebIconDatabase 对象实例。

(5) 向 WebViewCore 发送 INITIALIZE 事件，并且将 this 指针作为消息内容传递。WebView

类主要负责处理 UI 相关的事件，而 WebViewCore 主要负责与 WebCore 库交互。在运行时期，UI 线程和 WebCore 数据处理线程是运行在两个独立的线程中的。WebCoreThread 线程接收到 INITIALIZE 线程后，会调用消息对象参数的 initialize 方法，而后唤醒阻塞的 WebViewCore Java 线程(该线程在 WebViewCore 的构造函数中被阻塞)。不同的 WebView 对象实例有不同的 WebViewCore 对象实例，因此通过消息的方式可以使得 UI 线程和 WebViewCore 线程解耦合。

WebCoreThread 的事件处理函数处理 INITIALIZE 消息时，调用的是不同 WebView 中 WebViewCore 实例的 initialize 方法。WebViewCore 类中的 initialize 方法中会创建 BrowserFrame 对象(该对象管理整个 Web 窗体，以及 frame 相关事件)，并且向 WebView 对象发送 WEBCORE_INITIALIZED_MSG_ID 消息。WebView 消息处理函数能够根据其参数来初始化指定的 WebViewCore 对象，并且能够更新 WebViewCore 的 Frame 缓冲。

2. 载入数据

(1) 载入网络数据。

在 Android 应用开发过程中，可以使用类 WebView 的 loadUrl 方法请求访问指定的 URL 网页数据。在 WebView 对象中保存着 WebViewCore 的引用，由于 WebView 属于 UI 线程，而 WebViewCore 属于后台线程，因此 WebView 对象的 loadUrl 被调用时，会通过消息的方式将 URL 信息传递给 WebViewCore 对象，该对象会调用成员变量 mBrowserFrame 的 loadUrl 方法，进而调用 WebKit 库完成数据的载入。

当载入网络数据时，此功能分别由 Java 层和 C 层共同完成，其中 Java 层负责完成用户交互、资源下载等操作，而 C 层主要完成数据分析(建立 DOM 树、分析页面元素等)操作。由于 UI 线程和 WebCore 线程运行在不同的两个线程中，因此，当用户请求访问网络资源时，通过消息的方式向 WebViewCore 对象发送载入资源请求。

在 Java 层的 WebKit 模块中，所有与资源载入相关的操作都是由 BrowserFrame 类中对应的方法完成的，这些方法是本地方法，会直接调用 WebCore 库的 C 层函数完成数据载入请求，以及资源分析等操作。C 层的 FrameLoader 类是浏览框架的资源载入器，该类负责检查访问策略以及向 Java 层发送下载资源请求等功能。在 FrameLoader 中，当用户请求网络资源时，经过一系列的策略检查后，会调用 FrameBridge 的 startLoadingResource 方法，该方法会回调 BrowserFrame(Java)类的 startLoadingResource 方法，完成网络数据的下载，然后类 BrowserFrame (Java)的方法 startLoadingResource 会返回一个 LoadListener 的对象，FrameLoader 会删除原有的 FrameLoader 对象，将 LoadListener 对象封装成 ResourceLoadHandler 对象，并且将其设置为新的 FrameLoader。至此，完成了一次资源访问请求，接下来，库 WebCore 会根据资源数据来分析和构建 DOM，以及构建相关的数据结构。

(2) 载入本地数据。

所谓本地数据，是指以“data://”开头的 URL，载入本地数据的过程与载入网络数据的方法一样，只不过在执行 FrameLoader 类的 executeLoad 方法时，会根据 URL 的 SCHEME 类型区分，调用 DataLoader 的 requestUrl 方法，而不是调用 handleHTTPLoad 建立实际的网络通信连接。

(3) 载入文件数据。

所谓文件数据，是指以“file://”开头的 URL，载入的基本流程与网络数据载入流程基本

一致，不同的是，在运行 FrameLoader 类的 executeLoad 方法时，根据 SCHEME 类型，调用 FileLoader 的 requestUrl 方法来完成数据加载。

3. 刷新绘制

当用户拖动滚动条、有窗口遮盖，或者有页面事件触发时，都会向 WebViewCore(Java 层)对象发送背景重绘消息，该消息会引起网页数据的绘制操作。WebKit 的数据绘制可能出于效率上的考虑，没有通过 Java 层，而是直接在 C 层使用 SGL 库来完成。与 Java 层图形绘制相关的 Java 对象有 3 个，具体说明如下所示。

(1) Picture 类。

该类对 SGL 封装，其中变量 mNativePicture 实际上保存着 SkPicture 对象的指针。

WebViewCore 中定义了两个 Picture 对象，当作双缓冲处理，在调用 webKitDraw 方法时，会交换两个缓冲区，以加速刷新速度。

(2) WebView 类。

该类接受用户交互相关的操作，当有滚屏、窗口遮盖、用户点击页面按钮等相关操作时，WebView 对象会向与之相关的 WebViewCore 对象发送 VIEW_SIZE_CHANGED 消息。当 WebViewCore 对象接收到该消息后，将把构建时建立的 mContentPictureB 刷新到屏幕上，然后将 mContentPictureA 与之交换。

(3) WebViewCore 类。

该类封装了 WebKit 的 C 层代码，为视图类提供对 WebKit 的操作接口，所有对 WebKit 库的用户请求均由该类处理，该类还为视图类提供了两个 Picture 对象，用于图形数据刷新。

例如，我们拖拽 Web 页面，当用户使用手指点击触摸屏并且移动手指时，会引发 touch 事件，Android 平台会将 touch 事件传递给最前端的视图响应(dispatchTouchEvent 处理方法)。

在 WebView 类中，定义了 5 种 touch 模式，在手指拖动 Web 页面的情况下，会触发 mMotionDragMode，并且会调用 View 类的 scrollBy 方法，触发滚屏事件以及使视图无效(重绘，会调用 View 的 onDraw 方法)。WebView 视图中的滚屏事件由 onScrollChanged 方法响应，该方法向 WebViewCore 对象发送 SET_VISIBLE_RECT 事件。

WebViewCore 对象接收到 SET_VISIBLE_RECT 事件后，将消息参数中保存的新视图的矩形区域大小传递给 nativeSetVisibleRect 方法，通知 WebCoreViewImpl 对象(C 层)视图矩形变更(WebCoreViewImpl::setVisibleRect 方法)。

在 setVisibleRect 方法中，会通过虚拟机调用 WebViewCore 的 contentInvalidate 方法，该方法会引发 webkitDraw 方法的调用(通过 WEBKIT_DRAW 消息)。在方法 webkitDraw 中，首先会将 mContentPictureB 对象传递给本地方法 nativeDraw 绘制，然后将 mContentPictureB 的内容与 mContentPictureA 的内容互换。在这里，mContentPictureA 缓冲区是供给 WebViewCore 的 draw 方法使用的，如果用户选择某个控件，绘制焦点框的时候，WebViewCore 对象的 draw 方法会调用，绘制的内容保存在 mContentPictureA 中，之后会通过 Canvas 对象(Java 层)的 drawPicture 方法将其绘制到屏幕上，而 mContentPictureB 缓冲区是用于 built 操作的，nativeDraw 方法中首先会将传递的 mContentPictureB 对象数据重置，而后在重新构建的 mContentPictureB 画布上，将层上相关的元素绘制到该画布上。然后将 mContentPictureB 和 mContentPictureA 的内容互换，这样一次重绘事件产生时(会调用 WebView.onDraw 方法)会将 mContentPictureA 的

数据使用 Canvas 类的 drawPicture 方法绘制到屏幕上。当 webkitDraw 方法将 mContentPictureA 与 mContentPictureB 指针对调后，会向 WebView 对象发送 NEW_PICTURE_MSG_ID 消息，该消息会引发 WebViewCore 的 VIEW_SIZE_CHANGED 消息的产生，并且会使当前视图无效，产生重绘事件(invalidate())，引发 onDraw 方法的调用，完成一次网页数据的绘制过程。

15.1.4 WebView详解

在 Android 4.3 系统中，文件 WebView.java 是一个内置的支持浏览器的视图 View，此文件位于如下所示的目录中：

```
frameworks\base\core\java\android\webkit
```

在上述目录下，文件 WebView.java 实现了类 WebView，此类是 WebKit 模块 Java 层的视图类，所有需要使用 Web 浏览功能的 Android 应用程序都要创建该视图对象显示和处理请求的网络资源。目前，WebKit 模块支持 HTTP、HTTPS、FTP 以及 JavaScript 请求。WebView 作为应用程序的 UI 接口，为用户提供了一系列的网页浏览、用户交互接口，客户程序通过这些接口访问 WebKit 核心代码。

(1) 类 WebView 的构造函数。

类 WebView 继承于 AbsoluteLayout，实现 OnGlobalFocusChangeListener 和 OnHierarchy-ChangeListener，类 WebView 的构造函数的具体实现代码如下所示：

```
protected WebView(Context context, AttributeSet attrs, int defStyle,
  Map<String, Object> javaScriptInterfaces, boolean privateBrowsing) {
    super(context, attrs, defStyle);
    if (context == null) {
        throw new IllegalArgumentException("Invalid context argument");
    }
    sEnforceThreadChecking = context.getApplicationInfo().targetSdkVersion >=
      Build.VERSION_CODES.JELLY_BEAN_MR2;
    checkThread();

    ensureProviderCreated();
    mProvider.init(javaScriptInterfaces, privateBrowsing);
}
```

(2) 公共方法。

在文件 WebView.java 中提供了如下所示的公共方法：

```
//注入所提供的 Java 对象到这个 Web 视图
public void addJavascriptInterface(Object object, String name) {
    checkThread();
    mProvider.addJavascriptInterface(object, name);
}
//获取此 WebView 是否有后退历史的项目
public boolean canGoBack() {
    checkThread();
    return mProvider.canGoBack();
```

```
}
//获取页面是否可以返回或前进的步数量
public boolean canGoBackOrForward(int steps) {
    checkThread();
    return mProvider.canGoBackOrForward(steps);
}
//获取此 WebView 是否有历史前进的项目
public boolean canGoForward() {
    checkThread();
    return mProvider.canGoForward();
}
//获取此 WebView 是否可以将其放大
public boolean canZoomIn() {
    checkThread();
    return mProvider.canZoomIn();
}
//告诉这个 Web 视图，以清除其内部的前进/后退清单
public void clearHistory() {
    checkThread();
    mProvider.clearHistory();
}
//获取第一个子串组成的物理位置的地址
public static String findAddress(String addr) {
    return getFactory().getStatics().findAddress(addr);
}
//getContentHeight
//获取 HTML 内容的高度
public int getContentHeight() {
    checkThread();
    return mProvider.getContentHeight();
}
//获取当前页面的原始 URL
public String getOriginalUrl() {
    checkThread();
    return mProvider.getOriginalUrl();
}
//获取当前页面的进度
public int getProgress() {
    checkThread();
    return mProvider.getProgress();
}
```

15.1.5　WebViewCore详解

在 Android 4.3 系统中，文件 WebViewCore.java 位于如下所示的目录中：

```
frameworks\base\core\java\android\webkit
```

类 WebViewCore 是 Java 层与 C 层 WebKit 核心库的交互类，客户程序调用 WebView 的网

页浏览相关操作，会转发给 BrowserFrame 对象。当 WebKit 核心库完成实际的数据分析和处理后，会回调 WebViweCore 中定义的一系列 JNI 接口，这些接口会通过 CallbackProxy 将相关事件通知相应的 UI 对象。文件 WebViewCore.java 的主要实现代码如下所示：

```
private void initialize() {
    /* Initialize our private BrowserFrame class to handle all
    * frame-related functions. We need to create a new view which
    * in turn creates a C level FrameView and attaches it to the frame.
    */
    mBrowserFrame = new BrowserFrame(mContext, this, mCallbackProxy,
              mSettings, mJavascriptInterfaces);
    mJavascriptInterfaces = null;
    // Sync the native settings and also create the WebCore thread handler.
    mSettings.syncSettingsAndCreateHandler(mBrowserFrame);
    // Create the handler and transfer messages for the IconDatabase
    WebIconDatabaseClassic.getInstance().createHandler();
    // Create the handler for WebStorageClassic
    WebStorageClassic.getInstance().createHandler();
    // Create the handler for GeolocationPermissions.
    GeolocationPermissionsClassic.getInstance().createHandler();
    // The transferMessages call will transfer all pending messages to the
    // WebCore thread handler.
    mEventHub.transferMessages();

    // Send a message back to WebView to tell it that we have set up the
    // WebCore thread.
    if (mWebViewClassic != null) {
        Message.obtain(mWebViewClassic.mPrivateHandler,
                 WebViewClassic.WEBCORE_INITIALIZED_MSG_ID,
                 mNativeClass, 0).sendToTarget();
    }
}
private int mapDirection(int webkitDirection) {
    /*
    * This is WebKit's FocusDirection enum (from FocusDirection.h)
    enum FocusDirection {
        FocusDirectionNone = 0,
        FocusDirectionForward,
        FocusDirectionBackward,
        FocusDirectionUp,
        FocusDirectionDown,
        FocusDirectionLeft,
        FocusDirectionRight
    };
    */
    switch (webkitDirection) {
    case 1:
        return View.FOCUS_FORWARD;
```

```
    case 2:
        return View.FOCUS BACKWARD;
    case 3:
        return View.FOCUS_UP;
    case 4:
        return View.FOCUS_DOWN;
    case 5:
        return View.FOCUS_LEFT;
    case 6:
        return View.FOCUS_RIGHT;
    }
    return 0;
}
private String openFileChooser(String acceptType, String capture) {
    Uri uri = mCallbackProxy.openFileChooser(acceptType, capture);
    if (uri != null) {
        String filePath = "";
        // Note - querying for MediaStore.Images.Media.DATA
        // seems to work for all content URIs, not just images
        Cursor cursor = mContext.getContentResolver().query(
                    uri,
                    new String[] { MediaStore.Images.Media.DATA },
                    null, null, null);
        if (cursor != null) {
            try {
                if (cursor.moveToNext()) {
                    filePath = cursor.getString(0);
                }
            } finally {
                cursor.close();
            }
        } else {
            filePath = uri.getLastPathSegment();
        }
        String uriString = uri.toString();
        BrowserFrame.sJavaBridge.storeFilePathForContentUri(
          filePath, uriString);
        return uriString;
    }
    return "";
}
protected void populateVisitedLinks() {
    ValueCallback callback = new ValueCallback<String[]>() {
        @Override
        public void onReceiveValue(String[] value) {
            sendMessage(EventHub.POPULATE_VISITED_LINKS, (Object)value);
        }
    };
    mCallbackProxy.getVisitedHistory(callback);
```

```
}
private static class WebCoreThread implements Runnable {
    // Message id for initializing a new WebViewCore.
    private static final int INITIALIZE = 0;
    private static final int REDUCE_PRIORITY = 1;
    private static final int RESUME_PRIORITY = 2;

    @Override
    public void run() {
        Looper.prepare();
        Assert.assertNull(sWebCoreHandler);
        synchronized(WebViewCore.class) {
            sWebCoreHandler = new Handler() {
                @Override
                public void handleMessage(Message msg) {
                    switch (msg.what) {
                        case INITIALIZE:
                            WebViewCore core = (WebViewCore) msg.obj;
                            core.initialize();
                            break;

                        case REDUCE_PRIORITY:
                            // 3 is an adjustable number.
                            Process.setThreadPriority(
                                    Process.THREAD_PRIORITY_DEFAULT + 3 *
                                    Process.THREAD_PRIORITY_LESS_FAVORABLE);
                            break;

                        case RESUME_PRIORITY:
                            Process.setThreadPriority(
                                    Process.THREAD_PRIORITY_DEFAULT);
                            break;

                        case EventHub.ADD_PACKAGE_NAME:
                            if (BrowserFrame.sJavaBridge == null) {
                                throw new IllegalStateException(
                                  "No WebView has been created in this process!");
                            }
                            BrowserFrame.sJavaBridge.addPackageName(
                              (String)msg.obj);
                            break;

                        case EventHub.REMOVE_PACKAGE_NAME:
                            if (BrowserFrame.sJavaBridge == null) {
                                throw new IllegalStateException(
                                   "No WebView has been created in this process!");
                            }
                            BrowserFrame.sJavaBridge.removePackageName(
                              (String)msg.obj);
```

```
                        break;

                    case EventHub.PROXY_CHANGED:
                        if (BrowserFrame.sJavaBridge == null) {
                            throw new IllegalStateException(
                              "No WebView has been created in this process!");
                        }
                        BrowserFrame.sJavaBridge
                          .updateProxy((ProxyProperties)msg.obj);
                        break;

                    case EventHub.HEARTBEAT:
                        // Ping back the watchdog to let it know
                        // we're still processing
                        // messages.
                        Message m = (Message)msg.obj;
                        m.sendToTarget();
                        break;
                    case EventHub.TRUST_STORAGE_UPDATED:
                        // post a task to network thread for updating trust manager
                        nativeCertTrustChanged();
                        CertificateChainValidator.handleTrustStorageUpdate();
                        break;
                }
            }
        };
        WebViewCore.class.notify();
    }
    Looper.loop();
  }
}
static final String[] HandlerDebugString = {
    "REVEAL_SELECTION", // 96
    "", // 97
    "", // = 98
    "SCROLL_TEXT_INPUT", // = 99
    "LOAD_URL", // = 100;
    "STOP_LOADING", // = 101;
    "RELOAD", // = 102;
    "KEY_DOWN", // = 103;
    "KEY_UP", // = 104;
    "VIEW_SIZE_CHANGED", // = 105;
    "GO_BACK_FORWARD", // = 106;
    "SET_SCROLL_OFFSET", // = 107;
    "RESTORE_STATE", // = 108;
    "PAUSE_TIMERS", // = 109;
    "RESUME_TIMERS", // = 110;
    "CLEAR_CACHE", // = 111;
    "CLEAR_HISTORY", // = 112;
```

```
        "SET_SELECTION", // = 113;
        "REPLACE TEXT", // = 114;
        "PASS_TO_JS", // = 115;
        "SET_GLOBAL_BOUNDS", // = 116;
        "", // = 117;
        "CLICK", // = 118;
        "SET_NETWORK_STATE", // = 119;
        "DOC_HAS_IMAGES", // = 120;
        "FAKE_CLICK", // = 121;
        "DELETE_SELECTION", // = 122;
        "LISTBOX_CHOICES", // = 123;
        "SINGLE_LISTBOX_CHOICE", // = 124;
        "MESSAGE_RELAY", // = 125;
        "SET_BACKGROUND_COLOR", // = 126;
        "SET_MOVE_FOCUS", // = 127
        "SAVE_DOCUMENT_STATE", // = 128;
        "129", // = 129;
        "WEBKIT_DRAW", // = 130;
        "131", // = 131;
        "POST_URL", // = 132;
        "", // = 133;
        "CLEAR_CONTENT", // = 134;
        "", // = 135;
        "", // = 136;
        "REQUEST_CURSOR_HREF", // = 137;
        "ADD_JS_INTERFACE", // = 138;
        "LOAD_DATA", // = 139;
        "", // = 140;
        "", // = 141;
        "SET_ACTIVE", // = 142;
        "ON_PAUSE",     // = 143
        "ON_RESUME",    // = 144
        "FREE_MEMORY",  // = 145
        "VALID_NODE_BOUNDS", // = 146
        "SAVE_WEBARCHIVE", // = 147
        "WEBKIT_DRAW_LAYERS", // = 148;
        "REMOVE_JS_INTERFACE", // = 149;
    };
```

与文件 WebViewCore.java 相关的 C 类是 WebViewCoreI，在此类中定义了两个数据结构，一个是 WebViewCoreFields，对应于 Java 层 WebViewCore 对象的成员变量；另一个是 WebViewCore::JavaGlue，对应于 Java 层 WebViewCore 对象的成员方法。

具体的定义代码如下所示：

```
struct WebViewCoreFields {
   jfieldID m_nativeClass;
   jfieldID m_viewportWidth;
   jfieldID m_viewportHeight;
   jfieldID m_viewportInitialScale;
```

```
    jfieldID m_viewportMinimumScale;
    jfieldID m_viewportMaximumScale;
    jfieldID m_viewportUserScalable;
    jfieldID m_viewportDensityDpi;
    jfieldID m_webView;
    jfieldID m_drawIsPaused;
    jfieldID m_lowMemoryUsageMb;
    jfieldID m_highMemoryUsageMb;
    jfieldID m_highUsageDeltaMb;
} gWebViewCoreFields;

// ----------------------------

struct WebViewCore::JavaGlue {
    jweak m_obj;
    jmethodID m_scrollTo;
    jmethodID m_contentDraw;
    jmethodID m_layersDraw;
    jmethodID m_requestListBox;
    jmethodID m_openFileChooser;
    jmethodID m_requestSingleListBox;
    jmethodID m_jsAlert;
    jmethodID m_jsConfirm;
    jmethodID m_jsPrompt;
    jmethodID m_jsUnload;
    jmethodID m_jsInterrupt;
    jmethodID m_didFirstLayout;
    jmethodID m_updateViewport;
    jmethodID m_sendNotifyProgressFinished;
    jmethodID m_sendViewInvalidate;
    jmethodID m_updateTextfield;
    jmethodID m_updateTextSelection;
    jmethodID m_clearTextEntry;
    jmethodID m_restoreScale;
    jmethodID m_needTouchEvents;
    jmethodID m_requestKeyboard;
    jmethodID m_requestKeyboardWithSelection;
    jmethodID m_exceededDatabaseQuota;
    jmethodID m_reachedMaxAppCacheSize;
    jmethodID m_populateVisitedLinks;
    jmethodID m_geolocationPermissionsShowPrompt;
    jmethodID m_geolocationPermissionsHidePrompt;
    jmethodID m_getDeviceMotionService;
    jmethodID m_getDeviceOrientationService;
    jmethodID m_addMessageToConsole;
    jmethodID m_formDidBlur;
    jmethodID m_getPluginClass;
    jmethodID m_showFullScreenPlugin;
    jmethodID m_hideFullScreenPlugin;
```

```
    jmethodID m_createSurface;
    jmethodID m_addSurface;
    jmethodID m_updateSurface;
    jmethodID m_destroySurface;
    jmethodID m_getContext;
    jmethodID m_keepScreenOn;
    jmethodID m_sendFindAgain;
    jmethodID m_showRect;
    jmethodID m_centerFitRect;
    jmethodID m_setScrollbarModes;
    jmethodID m_setInstallableWebApp;
    jmethodID m_enterFullscreenForVideoLayer;
    jmethodID m_setWebTextViewAutoFillable;
    jmethodID m_selectAt;
    AutoJObject object(JNIEnv *env) {
        // We hold a weak reference to the Java WebViewCore to avoid memeory
        // leaks due to circular references when WebView.destroy() is not
        // called manually. The WebView and hence the WebViewCore could become
        // weakly reachable at any time, after which the GC could null our weak
        // reference, so we have to check the return value of this method at
        // every use. Note that our weak reference will be nulled before the
        // WebViewCore is finalized.
        return getRealObject(env, m_obj);
    }
};
```

在类 WebViewCore 中有一个作为成员变量的对象：JavaGlue，在构建 WebViewCore 对象时，用 WebViewCore(Java 层)中的方法 ID 值初始化该成员变量，并且会将构建的 WebViewCore 对象指针复制给 WebViewCore(Java 层)的 mNativeClass，这样就将 WebViewCore(Java 层)和 WebViewCore(C 层)关联起来。

15.2 Wi-Fi系统应用

Wi-Fi 是一种可以将个人电脑、手持设备(如 PDA、手机)等终端以无线方式互相连接的技术。Wi-Fi 是一个无线网路通信技术的品牌，由 Wi-Fi 联盟(Wi-Fi Alliance)所持有。目的是改善基于 IEEE 802.11 标准的无线网路产品之间的互通性。现在，一般人会把 Wi-Fi 及 IEEE 802.11 混为一谈。甚至直接把 Wi-Fi 等同于无线网际网路。在本节的内容中，将简要分析 Android 4.3 系统中 Wi-Fi 模块的基本源码。

15.2.1 Wi-Fi概述

在 Android 系统中，存在一个无线控制模块。打开方式如下：依次选择 Menu | Settings | Wireless$networks | Mobile network settings，打开如图 15-3 所示的界面，在此界面中，可以选择一个移动网络。

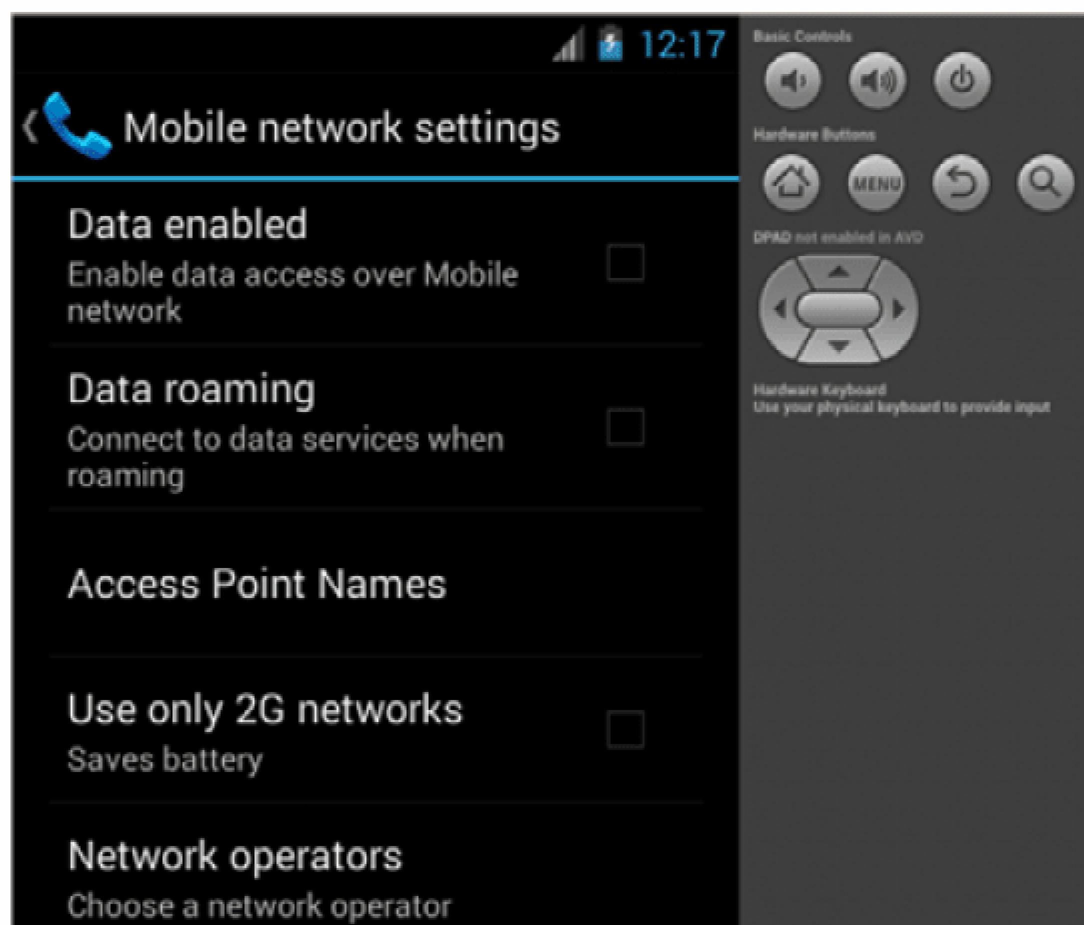


图 15-3　在此可以选择一个移动网络

15.2.2　Wi-Fi系统的层次结构

Wi-Fi 系统的上层接口包括数据部分和控制部分，数据部分通常是一个与以太网卡类似的网络设备，控制部分用于实现接入点操作和安全验证处理。

在软件层，Wi-Fi 系统包括 Linux 内核程序和协议，还包括本地部分、Java 框架类。Wi-Fi 系统向 Java 应用程序层提供了控制类的接口。

Android 平台中，Wi-Fi 系统的基本层次结构如图 15-4 所示。

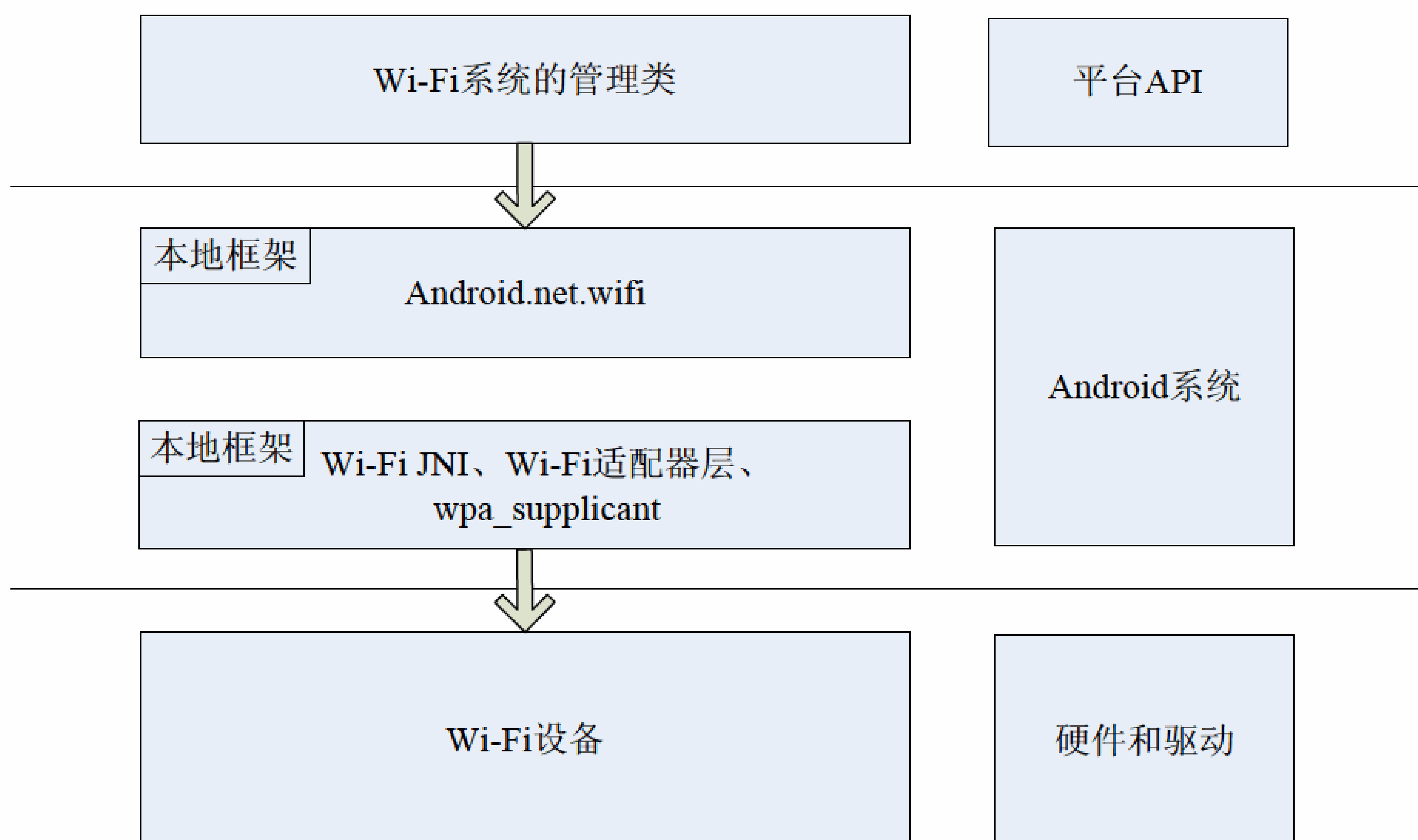


图 15-4　Wi-Fi系统的层次结构

由图 15-4 可知，Android 平台中 Wi-Fi 系统从上到下主要包括 Java 框架类、Android 适配器库、wpa_supplicant 守护进程、驱动程序和协议，这几部分的系统结构如图 15-5 所示。



图 15-5　Wi-Fi的系统结构

图 15-5 中，各个部分的具体说明如下所示。

(1)　Wi-Fi 用户空间的程序和库，对应路径如下所示：

```
external/wpa_supplicant/
```

在此生成库 libwpaclient.so 和守护进程 wpa_supplicant。

(2)　Wi-Fi 管理库，即适配器库，通过调用库 libwpaclient.so 成为 wpa_supplicant 在 Android 中的客户端。对应路径如下所示：

```
hardware/libhardware_legary/wifi/
```

(3)　JNI 部分的对应路径如下所示：

```
frameworks/base/core/jni/android_net_wifi_Wifi.cpp
```

(4)　Java 框架部分的对应路径如下所示：

```
frameworks/base/services/java/com/android/server/
frameworks/base/wifi/java/android/net/wifi/
```

在 android.net.wifi 将作为 Android 平台的 API 供 Java 应用程序层使用。

(5)　Wi-Fi Settings 应用程序的对应路径如下所示：

```
packages/apps/Settings/src/com/android/settings/wifi/
```

15.2.3　与Linux的差异

我们先看 Wi-Fi 在 Android 中是如何工作的：Android 使用一个修改版 wpa_supplicant 作为 daemon 来控制 Wi-Fi，代码位于如下目录中：

```
external/wpa_supplicant
```

wpa_supplicant 是通过 socket 与文件 hardware/libhardware_legacy/wifi/wifi.c 进行通信的。UI 通过 android.net.wifi 包(frameworks/base/wifi/java/android/net/wifi/)发送命令给文件 wifi.c。相应的 JNI 实现位于文件 frameworks/base/core/jni/android_net_wifi_Wifi.cpp 中，更高一级的网络管理位于如下目录中：

```
frameworks/base/core/java/android/net
```

在 Android 中的无线局域网部分是标准的系统，并且针对特定的硬件平台，所以需要移植和改动的内容并不多。在 Linux 内核中有 Wi-Fi 的标准协议，不同硬件平台的差异仅仅体现在 Wi-Fi 芯片驱动程序上。

除了这些芯片级驱动的差异外，在 Android 中实现其他无线局域网部分的方法在 Linux 内核中已经给出了具体方法。

而在 Android 用户空间中，使用了标准的 wpa_supplicant 守护进程，这也是一个标准的实现，所以无须我们为 Wi-Fi 增加单独的硬件抽象层代码，只需进行简单的配置工作即可。

15.2.4　分析本地部分的源码

本地实现部分主要包括 wpa_supplicant 以及 wpa_supplicant 适配层。

WPA 是 Wi-Fi Protected Access 的缩写 ，中文含义为“Wi-Fi 网络安全存取”。WPA 是一种基于标准的可互操作的 WLAN 安全性增强解决方案，可大大增强现有以及未来无线局域网系统的数据保护和访问控制水平。

wpa_supplicant 适配层是通用的 wpa_supplicant 的封装，在 Android 中作为 Wi-Fi 部分的硬件抽象层来使用。wpa_supplicant 适配层主要用于封装与 wpa_supplicant 守护进程的通信，以提供给 Android 框架使用。它实现了加载、控制和消息监控等功能。

wpa_supplicant 适配层的头文件如下所示：

```
hardware/libhardware_legacy/include/hardware_legacy/wifi.h
```

wpa_supplicant 的标准结构如图 15-6 所示。

我们重点关注框图的下半部分，即 wpa_supplicant 是如何与 DRIVER 进行联系的。整个过程暂以 AP 发出 SCAN 命令为主线。

由于现在大部分 Wi-Fi DRIVER 都支持 wext，所以就假设我们的设备走的是 wext 这条线，其实，用 ndis 也一样，整个流程也差不多。

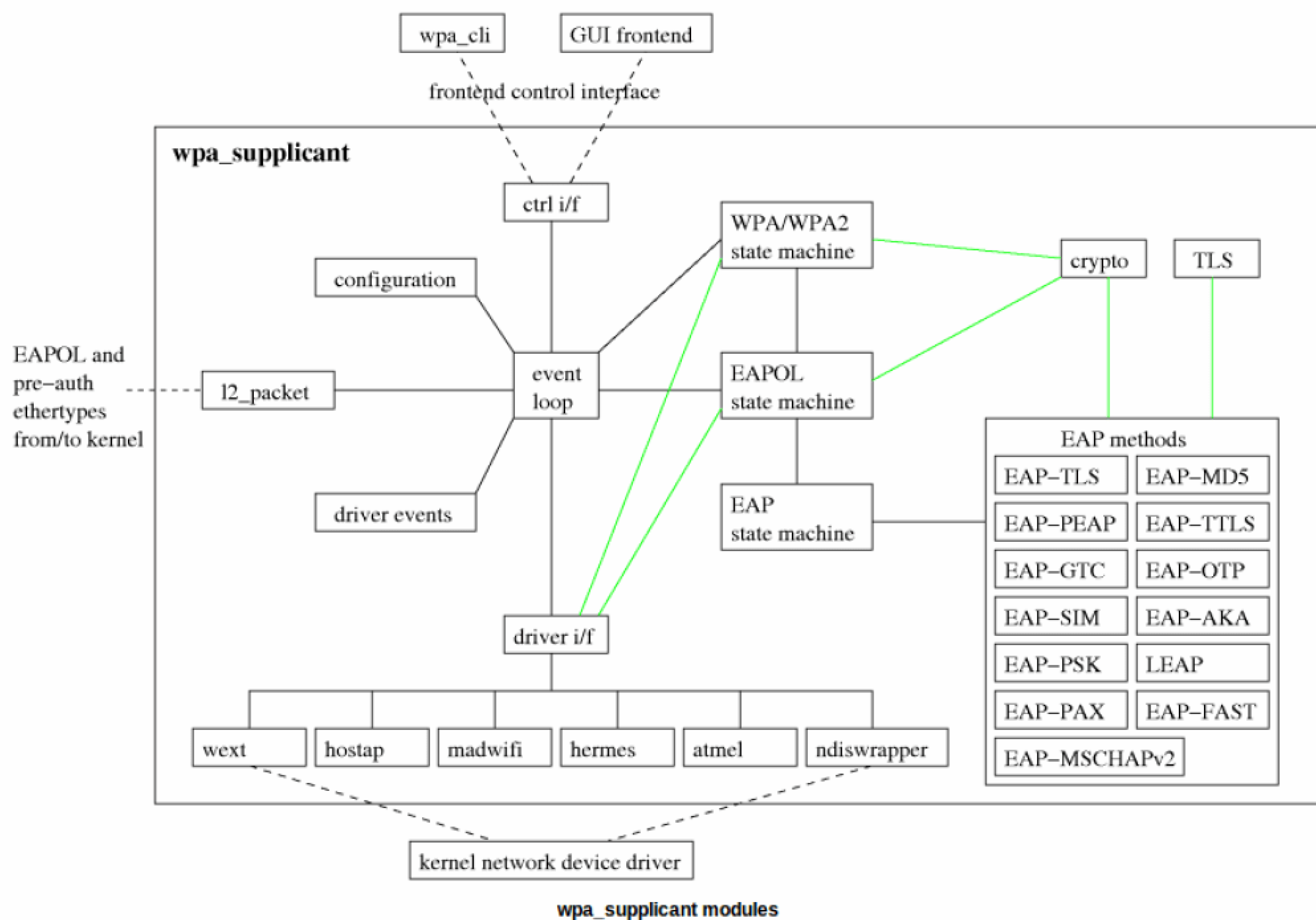


图 15-6　wpa_supplicant的标准结构

首先要说的是，在文件 Driver.h 文件中存在一个名为 wpa_driver_ops 的结构体，这个结构体在 Driver.c 中被声明如下代码：

```
#ifdef CONFIG_DRIVER_WEXT
extern struct wpa_driver_ops wpa_driver_wext_ops;
```

然后文件在 driver_wext.c 填写了该结构体的成员，代码如下所示：

```
const struct wpa_driver_ops wpa_driver_wext_ops = {
    .name = "wext",
    .desc = "Linux wireless extensions (generic)",
    .get_bssid = wpa_driver_wext_get_bssid,
    .get_ssid = wpa_driver_wext_get_ssid,
    .set_key = wpa_driver_wext_set_key,
    .set_countermeasures = wpa_driver_wext_set_countermeasures,
    .scan2 = wpa_driver_wext_scan,
    .get_scan_results2 = wpa_driver_wext_get_scan_results,
    .deauthenticate = wpa_driver_wext_deauthenticate,
    .associate = wpa_driver_wext_associate,
    .init = wpa_driver_wext_init,
    .deinit = wpa_driver_wext_deinit,
    .add_pmkid = wpa_driver_wext_add_pmkid,
    .remove_pmkid = wpa_driver_wext_remove_pmkid,
    .flush_pmkid = wpa_driver_wext_flush_pmkid,
    .get_capa = wpa_driver_wext_get_capa,
    .set_operstate = wpa_driver_wext_set_operstate,
```

```
    .get_radio_name = wext_get_radio_name,
#ifdef ANDROID
    .sched_scan = wext_sched_scan,
    .stop_sched_scan = wext_stop_sched_scan,
#endif /* ANDROID */
};
```

上述成员其实都是驱动和 wpa_supplicant 的接口，以 SCAN 为例的代码如下所示：

```
int wpa_driver_wext_scan(void *priv, const u8 *ssid, size_t ssid_len)
```

通过如下代码可以看出，wpa_cupplicant 是通过 IOCTL 来调用 SOCKET 与 DRIVER 进行通信的，并给 DRIVER 下达 SIOCSIWSCAN 这个命令：

```
if (ioctl(drv->ioctl_sock, SIOCSIWSCAN, &iwr) < 0)
```

这样，当一个命令从 AP 到 Framework 到 C++本地库再到 wpa_supplicant 适配层，再由 wpa_supplicant 下的 CMD 给 DRIVER 的路线就打通了。

因为 Wi-Fi 模块是采用 SDIO 总线来控制的，所以应该先记录下 CLIENT DRIVER 的 SDIO 部分的结构，此部分的 SDIO 分为三层，分别是 SdioDrv、SdioAdapter、SdioBusDrv。其中 SdioBusDrv 是 Client Driver 中 SDIO 与 WIFI 模块的接口，SdioAdapter 是 SdioDrv 和 SdioBusDrv 之间的适配层，SdioDrv 是 Client Driver 中 SDIO 与 Linux Kernel 中的 MMC SDIO 的接口。这三部分只需要关注一下 SdioDrv 就可以了，另外两层都只是对它的封装罢了。

在 SdioDrv 中提供了下面的功能：

```
static struct sdio_driver tiwlan_sdio_drv = {
    .probe          = tiwlan_sdio_probe,
    .remove         = tiwlan sdio remove,
    .name           = "sdio_tiwlan",
    .id_table       = tiwl12xx_devices,
};
int sdioDrv_EnableFunction(unsigned int uFunc);
int sdioDrv_EnableInterrupt(unsigned int uFunc);
```

Sdio 的读写实际上调用了 MMC\Core 中的如下功能函数：

```
static int mmc_io_rw_direct_host();
```

Sdio 功能部分读者只需简单了解即可，一般 HOST 部分芯片厂商都会提供完整的解决方案。我的主要任务还是 Wi-Fi 模块。

首先看 Wi-Fi 模块的入口函数 wlanDrvIf_ModuleInit()，此入口函数调用了函数 wlanDrvIf_Create()，主要代码如下所示：

```
static int wlanDrvIf_Create(void)
{
    TWlanDrvIfObj *drv; //这个结构体为代表设备，包含 Linux 网络设备结构体 net_device
    pDrvStaticHandle = drv;
    drv->pWorkQueue = create_singlethread_workqueue(TIWLAN_DRV_NAME); //创建了工作队列
    rc = wlanDrvIf_SetupNetif(drv);
    drv->wl_sock = netlink_kernel_create(NETLINK_USERSOCK, 0, NULL, NULL, THIS_MODULE);
```

```
    //创建了接受 wpa_supplicant 的 SOCKET 接口
    rc = drvMain_Create (drv,
                 &drv->tCommon.hDrvMain,
                 &drv->tCommon.hCmdHndlr,
                 &drv->tCommon.hContext,
                 &drv->tCommon.hTxDataQ,
                 &drv->tCommon.hTxMgmtQ,
                 &drv->tCommon.hTxCtrl,
                 &drv->tCommon.hTWD,
                 &drv->tCommon.hEvHandler,
                 &drv->tCommon.hCmdDispatch,
                 &drv->tCommon.hReport,
                 &drv->tCommon.hPwrState);
    rc = hPlatform_initInterrupt(drv, (void*)wlanDrvIf_HandleInterrupt);
    return 0;
}
```

在调用完函数 wlanDrvIf_Create()后，初始化 Wi-Fi 模块的工作就结束了。接下来开始分析如何实现初始化，首先分析函数 wlanDrvIf_SetupNetif(drv)，其主要实现代码如下所示：

```
static int wlanDrvIf_SetupNetif(TWlanDrvIfObj *drv)
{
   struct net_device *dev;
   int res;
   dev = alloc_etherdev (0); //开始申请 Linux 网络设备
   if (dev == NULL)
      ether_setup(dev); //开始建立网络接口，这两个都是 Linux 网络设备驱动的标准函数
   dev->netdev ops = &wlan netdev ops;
   wlanDrvWext_Init(dev);
   res = register_netdev(dev);
   hPlatform_SetupPm(wlanDrvIf_Suspend, wlanDrvIf_Resume, pDrvStaticHandle);
}
```

在此初始化了 wlanDrvWext_Inti(dev)，接下来需要注册网络设备 dev，在 wlan_netdev_ops 中的定义代码如下所示：

```
static const struct net_device_ops wlan_netdev_ops = {
      .ndo_open            = wlanDrvIf_Open,
      .ndo_stop            = wlanDrvIf_Release,
      .ndo_do_ioctl         = NULL,
      .ndo_start_xmit       = wlanDrvIf_Xmit,
      .ndo_get_stats        = wlanDrvIf_NetGetStat,
      .ndo_validate_addr     = NULL,
};
```

上述代码名字对应的都是 Linux 网络设备驱动的命令字，最后需要调用 rc=drvMain_CreateI，通过此函数完成了相关模块的初始化工作。

15.2.5 分析JNI部分的源码

在 Android 系统中，Wi-Fi 系统的 JNI 部分实现的源码文件如下：

```
frameworks/base/core/jni/android_net_wifi_Wifi.cpp
```

JNI 层的接口注册到 Java 层的源代码文件如下：

```
frameworks/base/wifi/java/android/net/wifi/WifiNative.java
```

WifiNative 将为 WifiService、WifiStateTracker、WifiMonitor 等几个 Wi-Fi 框架内部组件提供底层操作支持。

此处实现的本地函数都是通过调用 wpa_supplicant 适配层的接口来实现的(包含适配层的头文件 wifi.h)。wpa_supplicant 适配层是通用的 wpa_supplicant 的封装，在 Android 中作为 Wi-Fi 部分的硬件抽象层来使用。wpa_supplicant 适配层主要用于封装与 wpa_supplicant 守护进程的通信，以提供给 Android 框架使用。它实现了加载、控制和消息监控等功能。

wpa_supplicant 适配层的头文件如下所示：

```
hardware/libhardware_legacy/include/hardware_legacy/wifi.h
```

文件 wifi.h 是 Wi-Fi 适配器层对 JNI 部分的接口，在里面包含了一些加载和连接的控制接口，主要包括如下两个接口。

- wifi_command()：负责将命令发送到 Wi-Fi 下层。
- wifi_wait_for_event()：负责事件进入通道，此函数将被阻塞，直到收到一个 Wi-Fi 事件为止，并且以字符串的形式返回。

在文件 wifi.h 中定义上述接口的代码如下所示：

```
int wifi_command(const char *command, char *reply, size_t *reply_len);
int wifi_wait_for_event(char *buf, size_t len);
```

在文件 wifi.c 中实现了上述两个接口，具体代码如下所示：

```
int wifi_command(const char *command, char *reply, size_t *reply_len)
{
    return wifi_send_command(ctrl_conn, command, reply, reply_len);
}
int wifi_wait_for_event(char *buf, size_t buflen)
{
    size_t nread = buflen - 1;
    int fd;
    fd_set rfds;
    int result;
    struct timeval tval;
    struct timeval *tptr;

    if (monitor_conn == NULL)
        return 0;

    result = wpa_ctrl_recv(monitor_conn, buf, &nread);
    if (result < 0) {
        LOGD("wpa_ctrl_recv failed: %s\n", strerror(errno));
        return -1;
    }
```

```
    buf[nread] = '\0';
    if (result==0 && nread==0) {
        /* Fabricate an event to pass up */
        LOGD("Received EOF on supplicant socket\n");
        strncpy(buf, WPA_EVENT_TERMINATING, " - signal 0 received", buflen-1);
        buf[buflen-1] = '\0';
        return strlen(buf);
    }
    if (buf[0] == '<') {
        char *match = strchr(buf, '>');
        if (match != NULL) {
            nread -= (match+1-buf);
            memmove(buf, match+1, nread+1);
        }
    }
    return nread;
}
```

15.2.6 分析Java Framework部分的源码

Wi-Fi 系统 Java 部分的核心是根据 IWifiManager 接口所创建的 Binder 服务器端和客户端，服务器端是 WifiService，客户端是 WifiManager。具体结构如图 15-7 所示。

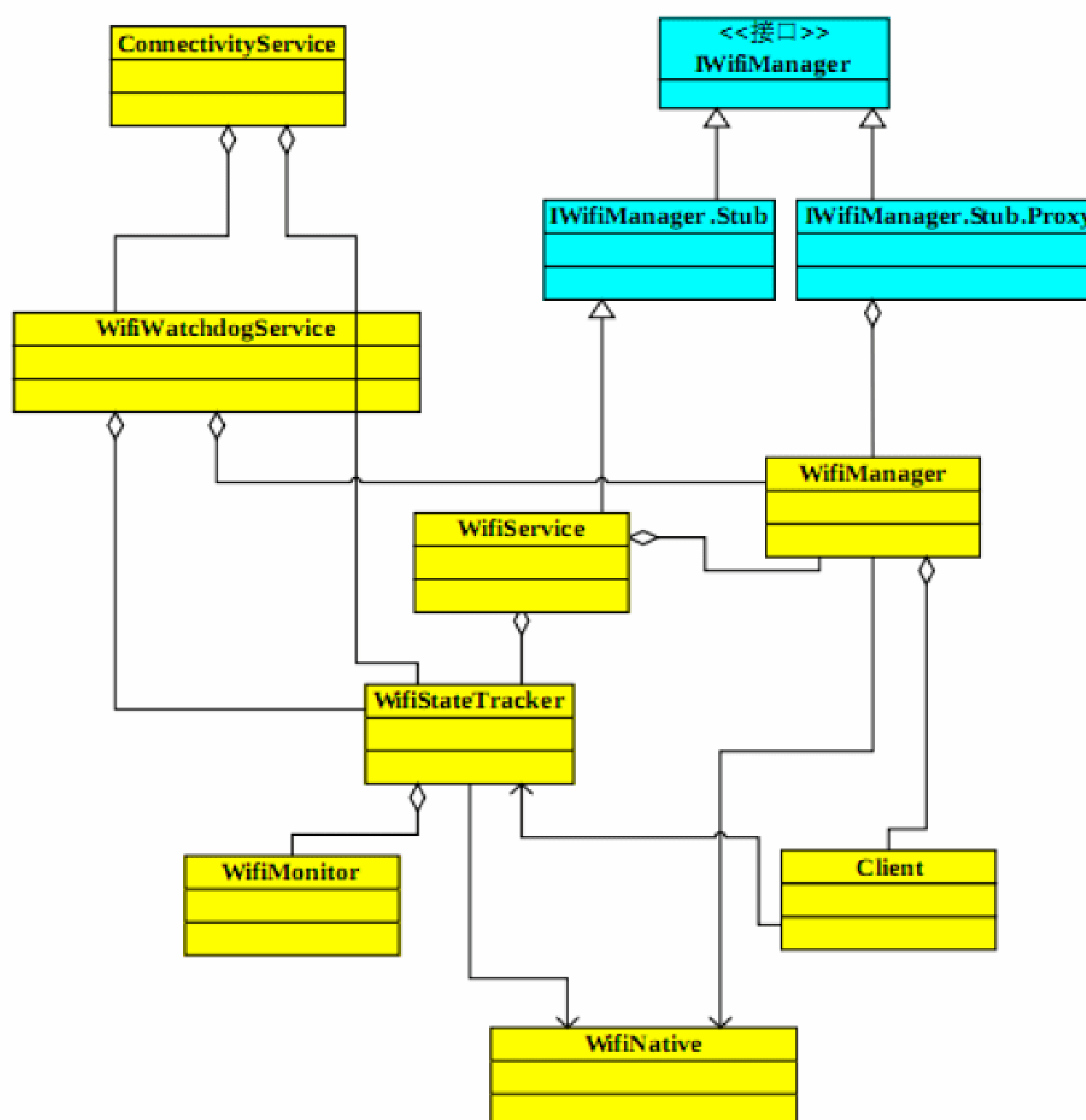


图 15-7 JNI接口的结构

Wi-Fi 系统的 Java 部分代码实现的目录如下所示：

```
frameworks/base/wifi/java/android/net/         //Wi-Fi 服务层的内容
frameworks/base/services/java/com/android/server/wifi   //Wi-Fi 部分的接口
```

Wi-Fi 系统 Java 层的核心是根据 IWifiManger 接口所创建的 Binder 服务器端和客户端，服务器端是 WifiService，客户端是 WifiManger。

编译 IWifiManger.aidl 生成文件 IWifiManger.java，并生成 IWifiManger.Stub(服务器端抽象类)和 IWifiManger.Stub.Proxy(客户端代理实现类)。WifiService 通过继承 IWifiManger.Stub 实现，而客户端通过 getService()函数获取 IWifiManger.Stub.Proxy(即 Service 的代理类)，将其作为参数传递给 WifiManger，供其与 WifiService 通信时使用。

图 15-7 中主要构成元素的具体说明如下所示。

(1)　WiFiManger

WiFiManger 部分表示 Wi-Fi 与外界的接口，用户通过它来访问 Wi-Fi 的核心功能。

WifiWatchdogService 这一系统组件也是用 WifiManger 来执行一些具体操作的。文件 WiFiManger.java 的主要实现代码如下所示：

```
public int addNetwork(WifiConfiguration config) {
    if (config == null) {
        return -1;
    }
    config.networkId = -1;
    return addOrUpdateNetwork(config);
}
public int updateNetwork(WifiConfiguration config) {
    if (config==null || config.networkId<0) {
        return -1;
    }
    return addOrUpdateNetwork(config);
}
private int addOrUpdateNetwork(WifiConfiguration config) {
    try {
        return mService.addOrUpdateNetwork(config);
    } catch (RemoteException e) {
        return -1;
    }
}
public boolean removeNetwork(int netId) {
    try {
        return mService.removeNetwork(netId);
    } catch (RemoteException e) {
        return false;
    }
}
public boolean enableNetwork(int netId, boolean disableOthers) {
    try {
        return mService.enableNetwork(netId, disableOthers);
```

```
    } catch (RemoteException e) {
        return false;
    }
}
public boolean disableNetwork(int netId) {
    try {
        return mService.disableNetwork(netId);
    } catch (RemoteException e) {
        return false;
    }
}
public boolean disconnect() {
    try {
        mService.disconnect();
        return true;
    } catch (RemoteException e) {
        return false;
    }
}
public boolean reconnect() {
    try {
        mService.reconnect();
        return true;
    } catch (RemoteException e) {
        return false;
    }
}
public boolean reassociate() {
    try {
        mService.reassociate();
        return true;
    } catch (RemoteException e) {
        return false;
    }
}
public boolean pingSupplicant() {
    if (mService == null)
        return false;
    try {
        return mService.pingSupplicant();
    } catch (RemoteException e) {
        return false;
    }
}
public boolean startScan() {
    try {
        mService.startScan();
        return true;
    } catch (RemoteException e) {
```

```
        return false;
    }
}
```

(2)　WifiService

WifiService 部分是服务器端的实现，作为 Wi-Fi 的核心，处理实际的驱动加载、扫描、链接、断开等命令，以及底层上报的事件。对于主动的命令控制，Wi-Fi 是一个简单的封装，针对来自客户端的控制命令，调用相应的 WifiNative 底层实现。

文件 WifiService.java 的主要实现代码如下所示：

```
public void handleMessage(Message msg) {
    switch (msg.what) {
        case AsyncChannel.CMD_CHANNEL_HALF_CONNECTED: {
            if (msg.arg1 == AsyncChannel.STATUS_SUCCESSFUL) {
                if (DBG)
                    Slog.d(TAG, "New client listening to asynchronous messages");
                // We track the clients by the Messenger
                // since it is expected to be always available
                mTrafficPoller.addClient(msg.replyTo);
            } else {
                Slog.e(TAG, "Client connection failure, error=" + msg.arg1);
            }
            break;
        }
        case AsyncChannel.CMD_CHANNEL_DISCONNECTED: {
            if (msg.arg1 == AsyncChannel.STATUS_SEND_UNSUCCESSFUL) {
                if (DBG) Slog.d(TAG, "Send failed, client connection lost");
            } else {
                if (DBG)
                  Slog.d(TAG, "Client connection lost with reason: " + msg.arg1);
            }
            mTrafficPoller.removeClient(msg.replyTo);
            break;
        }
        case AsyncChannel.CMD_CHANNEL_FULL_CONNECTION: {
            AsyncChannel ac = new AsyncChannel();
            ac.connect(mContext, this, msg.replyTo);
            break;
        }
        /* Client commands are forwarded to state machine */
        case WifiManager.CONNECT_NETWORK:
        case WifiManager.SAVE_NETWORK:
        case WifiManager.FORGET_NETWORK:
        case WifiManager.START_WPS:
        case WifiManager.CANCEL_WPS:
        case WifiManager.DISABLE_NETWORK:
        case WifiManager.RSSI_PKTCNT_FETCH: {
            mWifiStateMachine.sendMessage(Message.obtain(msg));
            break;
```

```
        }
        default: {
            Slog.d(TAG, "ClientHandler.handleMessage ignoring msg=" + msg);
            break;
        }
    }
}
private void noteScanStart() {
    WorkSource scanWorkSource = null;
    synchronized (WifiService.this) {
        if (mScanWorkSource != null) {
            // Scan already in progress, don't add this one to battery stats
            return;
        }
        scanWorkSource = new WorkSource(Binder.getCallingUid());
        mScanWorkSource = scanWorkSource;
    }

    long id = Binder.clearCallingIdentity();
    try {
        mBatteryStats.noteWifiScanStartedFromSource(scanWorkSource);
    } catch (RemoteException e) {
        Log.w(TAG, e);
    } finally {
        Binder.restoreCallingIdentity(id);
    }
}
private void noteScanEnd() {
    WorkSource scanWorkSource = null;
    synchronized (WifiService.this) {
        scanWorkSource = mScanWorkSource;
        mScanWorkSource = null;
    }
    if (scanWorkSource != null) {
        try {
            mBatteryStats.noteWifiScanStoppedFromSource(scanWorkSource);
        } catch (RemoteException e) {
            Log.w(TAG, e);
        }
    }
}
public void checkAndStartWifi() {
    /* Check if wi-fi needs to be enabled */
    boolean wifiEnabled = mSettingsStore.isWifiToggleEnabled();
    Slog.i(TAG, "WifiService starting up with Wi-Fi "
      + (wifiEnabled? "enabled" : "disabled"));

    // If we are already disabled (could be due to airplane mode),
    // avoid changing persist state here
```

```
    if (wifiEnabled) setWifiEnabled(wifiEnabled);

    mWifiWatchdogStateMachine = WifiWatchdogStateMachine.
    makeWifiWatchdogStateMachine(mContext);

}
public boolean removeNetwork(int netId) {
    enforceChangePermission();
    if (mWifiStateMachineChannel != null) {
        return mWifiStateMachine.syncRemoveNetwork(
          mWifiStateMachineChannel, netId);
    } else {
        Slog.e(TAG, "mWifiStateMachineChannel is not initialized");
        return false;
    }
}
public boolean enableNetwork(int netId, boolean disableOthers) {
    enforceChangePermission();
    if (mWifiStateMachineChannel != null) {
        return mWifiStateMachine.syncEnableNetwork(
          mWifiStateMachineChannel, netId, disableOthers);
    } else {
        Slog.e(TAG, "mWifiStateMachineChannel is not initialized");
        return false;
    }
}
```

当接收到客户端的命令后，一般会将其转换成对应的自身消息塞入消息队列中，以便客户端的调用可以及时返回，然后在 WifiHandler 的 handleMessage()中处理对应的消息。而底层上报的事件，WifiService 则通过启动 WifiStateTracker 来负责处理。WifiStateTracker 和 WifiMonitor 的具体功能如下所示。

① WifiStateTracker：除了负责 Wi-Fi 的电源管理模式等功能外，其核心是 WifiMonitor 所实现的事件轮询机制，及消息处理函数 handleMessage()。文件 WifiStateTracker.java 在 frameworks\base\wifi\java\android\net\wifi\目录中定义，主要实现代码如下所示：

```
public void startMonitoring(Context context, Handler target) {
    mCsHandler = target;
    mContext = context;

    mWifiManager =
      (WifiManager) mContext.getSystemService(Context.WIFI_SERVICE);
    IntentFilter filter = new IntentFilter();
    filter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    filter.addAction(WifiManager.LINK_CONFIGURATION_CHANGED_ACTION);

    mWifiStateReceiver = new WifiStateReceiver();
    mContext.registerReceiver(mWifiStateReceiver, filter);
}
```

```
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
        mNetworkInfo = (NetworkInfo)intent.getParcelableExtra(
          WifiManager.EXTRA_NETWORK_INFO);
        mLinkProperties = intent.getParcelableExtra(
          WifiManager.EXTRA_LINK_PROPERTIES);
        if (mLinkProperties == null) {
            mLinkProperties = new LinkProperties();
        }
        mLinkCapabilities = intent.getParcelableExtra(
          WifiManager.EXTRA_LINK_CAPABILITIES);
        if (mLinkCapabilities == null) {
            mLinkCapabilities = new LinkCapabilities();
        }
        // don't want to send redundent state messages
        // but send portal check detailed state notice
        NetworkInfo.State state = mNetworkInfo.getState();
        if (mLastState==state
          && mNetworkInfo.getDetailedState()
          !=DetailedState.CAPTIVE_PORTAL_CHECK) {
            return;
        } else {
            mLastState = state;
        }
        Message msg = mCsHandler.obtainMessage(EVENT_STATE_CHANGED,
          new NetworkInfo(mNetworkInfo));
        msg.sendToTarget();
    } else if (intent.getAction().equals(
      WifiManager.LINK_CONFIGURATION_CHANGED_ACTION)) {
        mLinkProperties = (LinkProperties)intent.getParcelableExtra(
          WifiManager.EXTRA_LINK_PROPERTIES);
        Message msg = mCsHandler.obtainMessage(
          EVENT_CONFIGURATION_CHANGED, mNetworkInfo);
        msg.sendToTarget();
    }
}
```

由此可见，WifiStateTracker 也是 Wi-Fi 部分与外界的接口，它不像 WifiManger 那样直接被实例化来操作，而是通过 Intent 机制来发消息通知给客户端注册的 BroadcastReceiver，以完成和客户端的接口。

② WifiMonitor：通过开启一个 MonitorThread 来实现事件的轮询，轮询的关键函数是前面提到的阻塞式函数 WifiNative.waitForEvent()。获取事件后，WifiMonitor 通过一系列的 Handler 通知给 WifiStateTracker。

这里 WifiMonitor 的通知机制是将底层事件转换成 WifiStateTracker 所能识别的消息，塞入 WifiStateTracker 的消息循环中，最终在 handleMessage()中由 WifiStateTracker 完成对应的处理。文件 WifiMonitor.java 在目录 frameworks\base\wifi\java\android\net\wifi\中定义，主要实现代码如下所示：

```
public class WifiMonitor {
    private static final String TAG = "WifiMonitor";
    private static final int CONNECTED    = 1;
    private static final int DISCONNECTED = 2;
    private static final int STATE_CHANGE = 3;
    private static final int SCAN_RESULTS = 4;
    private static final int LINK_SPEED   = 5;
    private static final int TERMINATING  = 6;
    private static final int DRIVER_STATE = 7;
    private static final int EAP_FAILURE  = 8;
    private static final int UNKNOWN      = 9;

    private static final String EVENT_PREFIX_STR = "CTRL-EVENT-";
    private static final int EVENT_PREFIX_LEN_STR = EVENT_PREFIX_STR.length();

    private static final String WPA_EVENT_PREFIX_STR = "WPA:";
    private static final String PASSWORD_MAY_BE_INCORRECT_STR =
      "pre-shared key may be incorrect";

    private static final String WPS_SUCCESS_STR = "WPS-SUCCESS";

    private static final String WPS_FAIL_STR    = "WPS-FAIL";
    private static final String WPS_FAIL_PATTERN =
      "WPS-FAIL msg=\\d+(?: config_error=(\\d+))?(?: reason=(\\d+))?";

    private static final int CONFIG_MULTIPLE_PBC_DETECTED = 12;
    private static final int CONFIG_AUTH_FAILURE = 18;

    /* reason code values for reason=%d */
    private static final int REASON_TKIP_ONLY_PROHIBITED = 1;
    private static final int REASON_WEP_PROHIBITED = 2;

    private static final String WPS_OVERLAP_STR = "WPS-OVERLAP-DETECTED";
    private static final String WPS_TIMEOUT_STR = "WPS-TIMEOUT";

    private static final String CONNECTED_STR =   "CONNECTED";
    private static final String DISCONNECTED_STR = "DISCONNECTED";
    private static final String STATE_CHANGE_STR =  "STATE-CHANGE";
    private static final String SCAN_RESULTS_STR =  "SCAN-RESULTS";
    private static final String LINK_SPEED_STR = "LINK-SPEED";
    private static final String TERMINATING_STR =  "TERMINATING";
    private static final String DRIVER_STATE_STR = "DRIVER-STATE";
    private static final String EAP_FAILURE_STR = "EAP-FAILURE";
    private static final String EAP_AUTH_FAILURE_STR = "EAP authentication failed";

    private static Pattern mConnectedEventPattern =
      Pattern.compile("((?:[0-9a-f]{2}:){5}[0-9a-f]{2}) .* \\[id=([0-9]+)");
```

```
private static final String P2P_EVENT_PREFIX_STR = "P2P";

private static final String P2P_DEVICE_FOUND_STR = "P2P-DEVICE-FOUND";
private static final String P2P_DEVICE_LOST_STR = "P2P-DEVICE-LOST";
private static final String P2P_FIND_STOPPED_STR = "P2P-FIND-STOPPED";
private static final String P2P_GO_NEG_REQUEST_STR = "P2P-GO-NEG-REQUEST";

private static final String P2P_GO_NEG_SUCCESS_STR = "P2P-GO-NEG-SUCCESS";
private static final String P2P_GO_NEG_FAILURE_STR = "P2P-GO-NEG-FAILURE";

private static final String P2P_GROUP_FORMATION_SUCCESS_STR =
  "P2P-GROUP-FORMATION-SUCCESS";

private static final String P2P_GROUP_FORMATION_FAILURE_STR =
  "P2P-GROUP-FORMATION-FAILURE";

private static final String P2P_GROUP_STARTED_STR = "P2P-GROUP-STARTED";
private static final String P2P_GROUP_REMOVED_STR = "P2P-GROUP-REMOVED";
private static final String P2P_INVITATION_RECEIVED_STR =
  "P2P-INVITATION-RECEIVED";
private static final String P2P_INVITATION_RESULT_STR =
  "P2P-INVITATION-RESULT";
private static final String P2P_PROV_DISC_PBC_REQ_STR =
  "P2P-PROV-DISC-PBC-REQ";
private static final String P2P_PROV_DISC_PBC_RSP_STR =
  "P2P-PROV-DISC-PBC-RESP";
private static final String P2P_PROV_DISC_ENTER_PIN_STR =
  "P2P-PROV-DISC-ENTER-PIN";
private static final String P2P_PROV_DISC_SHOW_PIN_STR =
  "P2P-PROV-DISC-SHOW-PIN";
private static final String P2P_PROV_DISC_FAILURE_STR =
  "P2P-PROV-DISC-FAILURE";
private static final String P2P_SERV_DISC_RESP_STR = "P2P-SERV-DISC-RESP";

private static final String HOST_AP_EVENT_PREFIX_STR = "AP";
private static final String AP_STA_CONNECTED_STR = "AP-STA-CONNECTED";
private static final String AP_STA_DISCONNECTED_STR = "AP-STA-DISCONNECTED";

private final StateMachine mStateMachine;
private final WifiNative mWifiNative;
private static final int BASE = Protocol.BASE_WIFI_MONITOR;
public static final int SUP_CONNECTION_EVENT                  = BASE + 1;
public static final int SUP_DISCONNECTION_EVENT               = BASE + 2;
public static final int NETWORK_CONNECTION_EVENT              = BASE + 3;
/* Network disconnection completed */
public static final int NETWORK_DISCONNECTION_EVENT           = BASE + 4;
/* Scan results are available */
public static final int SCAN_RESULTS_EVENT                    = BASE + 5;
/* Supplicate state changed */
```

```
public static final int SUPPLICANT_STATE_CHANGE_EVENT      = BASE + 6;
/* Password failure and EAP authentication failure */
public static final int AUTHENTICATION_FAILURE_EVENT       = BASE + 7;
/* WPS success detected */
public static final int WPS_SUCCESS_EVENT                 = BASE + 8;
/* WPS failure detected */
public static final int WPS_FAIL_EVENT                    = BASE + 9;
/* WPS overlap detected */
public static final int WPS_OVERLAP_EVENT                 = BASE + 10;
/* WPS timeout detected */
public static final int WPS_TIMEOUT_EVENT                 = BASE + 11;
/* Driver was hung */
public static final int DRIVER_HUNG_EVENT                 = BASE + 12;

/* P2P events */
public static final int P2P_DEVICE_FOUND_EVENT              = BASE + 21;
public static final int P2P_DEVICE_LOST_EVENT               = BASE + 22;
public static final int P2P_GO_NEGOTIATION_REQUEST_EVENT    = BASE + 23;
public static final int P2P_GO_NEGOTIATION_SUCCESS_EVENT    = BASE + 25;
public static final int P2P_GO_NEGOTIATION_FAILURE_EVENT    = BASE + 26;
public static final int P2P_GROUP_FORMATION_SUCCESS_EVENT   = BASE + 27;
public static final int P2P_GROUP_FORMATION_FAILURE_EVENT   = BASE + 28;
public static final int P2P_GROUP_STARTED_EVENT             = BASE + 29;
public static final int P2P_GROUP_REMOVED_EVENT             = BASE + 30;
public static final int P2P_INVITATION_RECEIVED_EVENT       = BASE + 31;
public static final int P2P_INVITATION_RESULT_EVENT         = BASE + 32;
public static final int P2P_PROV_DISC_PBC_REQ_EVENT         = BASE + 33;
public static final int P2P_PROV_DISC_PBC_RSP_EVENT         = BASE + 34;
public static final int P2P_PROV_DISC_ENTER_PIN_EVENT       = BASE + 35;
public static final int P2P_PROV_DISC_SHOW_PIN_EVENT        = BASE + 36;
public static final int P2P_FIND_STOPPED_EVENT              = BASE + 37;
public static final int P2P_SERV_DISC_RESP_EVENT            = BASE + 38;
public static final int P2P_PROV_DISC_FAILURE_EVENT         = BASE + 39;

/* hostap events */
public static final int AP_STA_DISCONNECTED_EVENT           = BASE + 41;
public static final int AP_STA_CONNECTED_EVENT              = BASE + 42;

private static final String MONITOR_SOCKET_CLOSED_STR = "connection closed";
private static final String WPA_RECV_ERROR_STR = "recv error";

private int mRecvErrors = 0;
private static final int MAX_RECV_ERRORS    = 10;

public WifiMonitor(StateMachine wifiStateMachine, WifiNative wifiNative) {
    mStateMachine = wifiStateMachine;
    mWifiNative = wifiNative;
}
```

```
public void startMonitoring() {
    new MonitorThread().start();
}
public void run() {
    if (connectToSupplicant()) {
        mStateMachine.sendMessage(SUP_CONNECTION_EVENT);
    } else {
        mStateMachine.sendMessage(SUP_DISCONNECTION_EVENT);
        return;
    }

    //noinspection InfiniteLoopStatement
    for (;;) {
        String eventStr = mWifiNative.waitForEvent();

        // Skip logging the common but mostly uninteresting scan-results event
        if (false && eventStr.indexOf(SCAN_RESULTS_STR)==-1) {
            Log.d(TAG, "Event [" + eventStr + "]");
        }
        if (!eventStr.startsWith(EVENT_PREFIX_STR)) {
            if (eventStr.startsWith(WPA_EVENT_PREFIX_STR)
              && 0<eventStr.indexOf(PASSWORD_MAY_BE_INCORRECT_STR)) {
                mStateMachine.sendMessage(AUTHENTICATION_FAILURE_EVENT);
            } else if (eventStr.startsWith(WPS_SUCCESS_STR)) {
                mStateMachine.sendMessage(WPS_SUCCESS_EVENT);
            } else if (eventStr.startsWith(WPS_FAIL_STR)) {
                handleWpsFailEvent(eventStr);
            } else if (eventStr.startsWith(WPS_OVERLAP_STR)) {
                mStateMachine.sendMessage(WPS_OVERLAP_EVENT);
            } else if (eventStr.startsWith(WPS_TIMEOUT_STR)) {
                mStateMachine.sendMessage(WPS_TIMEOUT_EVENT);
            } else if (eventStr.startsWith(P2P_EVENT_PREFIX_STR)) {
                handleP2pEvents(eventStr);
            } else if (eventStr.startsWith(HOST_AP_EVENT_PREFIX_STR)) {
                handleHostApEvents(eventStr);
            }
            continue;
        }
        String eventName = eventStr.substring(EVENT_PREFIX_LEN_STR);
        int nameEnd = eventName.indexOf(' ');
        if (nameEnd != -1)
            eventName = eventName.substring(0, nameEnd);
        if (eventName.length() == 0) {
            if (false)
              Log.i(TAG, "Received wpa_supplicant event with empty event name");
                continue;
        }
        /*
        * Map event name into event enum
```

```
        */
        int event;
        if (eventName.equals(CONNECTED_STR))
            event = CONNECTED;
        else if (eventName.equals(DISCONNECTED_STR))
            event = DISCONNECTED;
        else if (eventName.equals(STATE_CHANGE_STR))
            event = STATE_CHANGE;
        else if (eventName.equals(SCAN_RESULTS_STR))
            event = SCAN_RESULTS;
        else if (eventName.equals(LINK_SPEED_STR))
            event = LINK_SPEED;
        else if (eventName.equals(TERMINATING_STR))
            event = TERMINATING;
        else if (eventName.equals(DRIVER_STATE_STR))
            event = DRIVER_STATE;
        else if (eventName.equals(EAP_FAILURE_STR))
            event = EAP_FAILURE;
        else
            event = UNKNOWN;

        String eventData = eventStr;
        if (event == DRIVER_STATE || event == LINK_SPEED)
            eventData = eventData.split(" ")[1];
        else if (event==STATE_CHANGE || event==EAP_FAILURE) {
            int ind = eventStr.indexOf(" ");
            if (ind != -1) {
                eventData = eventStr.substring(ind + 1);
            }
        } else {
            int ind = eventStr.indexOf(" - ");
            if (ind != -1) {
                eventData = eventStr.substring(ind + 3);
            }
        }

        if (event == STATE_CHANGE) {
            handleSupplicantStateChange(eventData);
        } else if (event == DRIVER_STATE) {
            handleDriverEvent(eventData);
        } else if (event == TERMINATING) {
            /**
            * Close the supplicant connection if we see
            * too many recv errors
            */
            if (eventData.startsWith(WPA_RECV_ERROR_STR)) {
                if (++mRecvErrors > MAX_RECV_ERRORS) {
                    if (false) {
                        Log.d(TAG, "too many recv errors, closing connection");
```

```
                    }
                } else {
                    continue;
                }
            }

            // notify and exit
            mStateMachine.sendMessage(SUP_DISCONNECTION_EVENT);
            break;
        } else if (event == EAP_FAILURE) {
            if (eventData.startsWith(EAP_AUTH_FAILURE_STR)) {
                mStateMachine.sendMessage(AUTHENTICATION_FAILURE_EVENT);
            }
        } else {
            handleEvent(event, eventData);
        }
        mRecvErrors = 0;
    }
}

private boolean connectToSupplicant() {
    int connectTries = 0;
    while (true) {
        if (mWifiNative.connectToSupplicant()) {
            return true;
        }
        if (connectTries++ < 5) {
            nap(1);
        } else {
            break;
        }
    }
    return false;
}
...
```

(3)　WifiWatchdogService

此部分是 ConnectivityService 所启动的服务，但它并不是通过 Binder 来实现的服务。它的作用是监控同一个网络内的接入点(Access Point)，如果当前接入点的 DNS 无法 ping 通，就自动切换到下一个接入点。

WifiWatchdogService 通过 WifiManger 和 WifiStateTracker 辅助完成具体的控制动作。

在 WifiWatchdogService 初始化时，通过 registerForWifiBroadcasts 注册获取网络变化的 BroadcastReceiver，也就是捕获 WifiStateTracker 所发出的通知消息，并开启一个 WifiWatchdog-Thread 线程来处理获取的消息。通过更改 Setting.Secure.WIFI_WARCHDOG_ON 的配置，可以开启和关闭 WifiWatchdogService。

15.2.7　分析Setting中的设置部分的源码

Android 的 Settings 应用程序对 Wi-Fi 的使用，是典型的 Wi-Fi 应用方式，也是用户可见的 Android WiFi 管理程序。此部分源代码的目录如下所示：

```
packages/apps/Settings/src/com/android/settings/wifi/
```

Setting 里的 Wi-Fi 部分是用户可见的设置界面，提供 Wi-Fi 开关、扫描 AP、链接/断开的基本功能。其中 WifiEnabler 通过 WifiManger 来完成实际的功能，也同样注册一个 BroadcastReceiver 来响应 WifiStateTracker 所发出的通知消息。

WifiEnabler 其实是一个比较简单的类，提供开启和关闭 Wi-Fi 的功能。设置里面的外层 Wi-Fi 开关菜单，就是直接通过它来做到的。

文件 WifiEnabler.java 的具体实现代码如下所示：

```
public class WifiEnabler implements CompoundButton.OnCheckedChangeListener  {
    private final Context mContext;
    private Switch mSwitch;
    private AtomicBoolean mConnected = new AtomicBoolean(false);

    private final WifiManager mWifiManager;
    private boolean mStateMachineEvent;
    private final IntentFilter mIntentFilter;
    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (WifiManager.WIFI_STATE_CHANGED_ACTION.equals(action)) {
                handleWifiStateChanged(intent.getIntExtra(
                  WifiManager.EXTRA_WIFI_STATE, WifiManager.WIFI_STATE_UNKNOWN));
            } else if (WifiManager.SUPPLICANT_STATE_CHANGED_ACTION.equals(action)) {
                if (!mConnected.get()) {
                    handleStateChanged(WifiInfo.getDetailedStateOf(
                       (SupplicantState)intent.getParcelableExtra(
                        WifiManager.EXTRA_NEW_STATE)));
                }
            } else if (WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(action)) {
                NetworkInfo info = (NetworkInfo)intent.getParcelableExtra(
                   WifiManager.EXTRA_NETWORK_INFO);
                mConnected.set(info.isConnected());
                handleStateChanged(info.getDetailedState());
            }
        }
    };
    public WifiEnabler(Context context, Switch switch ) {
        mContext = context;
        mSwitch = switch_;
        mWifiManager = (WifiManager)context.getSystemService(Context.WIFI_SERVICE);
```

```
        mIntentFilter = new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);
        // The order matters! We really should not depend on this. :(
        mIntentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
        mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    }

    public void resume() {
        // Wi-Fi state is sticky, so just let the receiver update UI
        mContext.registerReceiver(mReceiver, mIntentFilter);
        mSwitch.setOnCheckedChangeListener(this);
    }

    public void pause() {
        mContext.unregisterReceiver(mReceiver);
        mSwitch.setOnCheckedChangeListener(null);
    }

    public void setSwitch(Switch switch_) {
        if (mSwitch == switch_) return;
        mSwitch.setOnCheckedChangeListener(null);
        mSwitch = switch ;
        mSwitch.setOnCheckedChangeListener(this);

        final int wifiState = mWifiManager.getWifiState();
        boolean isEnabled = wifiState == WifiManager.WIFI_STATE_ENABLED;
        boolean isDisabled = wifiState == WifiManager.WIFI_STATE_DISABLED;
        mSwitch.setChecked(isEnabled);
        mSwitch.setEnabled(isEnabled || isDisabled);
    }

    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        //Do nothing if called as a result of a state machine event
        if (mStateMachineEvent) {
            return;
        }
        // Show toast message if Wi-Fi is not allowed in airplane mode
        if (isChecked && !WirelessSettings.isRadioAllowed(mContext,
          Settings.Global.RADIO_WIFI)) {
            Toast.makeText(mContext, R.string.wifi_in_airplane_mode,
                           Toast.LENGTH_SHORT).show();
            // Reset switch to off. No infinite check/listenenr loop.
            buttonView.setChecked(false);
        }

        // Disable tethering if enabling Wifi
        int wifiApState = mWifiManager.getWifiApState();
        if (isChecked && ((wifiApState==WifiManager.WIFI_AP_STATE_ENABLING)
         || (wifiApState==WifiManager.WIFI AP STATE ENABLED))) {
            mWifiManager.setWifiApEnabled(null, false);
```

```
        }
        if (mWifiManager.setWifiEnabled(isChecked)) {
            // Intent has been taken into account, disable until new state is active
            mSwitch.setEnabled(false);
        } else {
            // Error
            Toast.makeText(mContext, R.string.wifi_error, Toast.LENGTH_SHORT).show();
        }
    }
    private void handleWifiStateChanged(int state) {
        switch (state) {
            case WifiManager.WIFI_STATE_ENABLING:
                mSwitch.setEnabled(false);
                break;
            case WifiManager.WIFI_STATE_ENABLED:
                setSwitchChecked(true);
                mSwitch.setEnabled(true);
                break;
            case WifiManager.WIFI_STATE_DISABLING:
                mSwitch.setEnabled(false);
                break;
            case WifiManager.WIFI_STATE_DISABLED:
                setSwitchChecked(false);
                mSwitch.setEnabled(true);
                break;
            default:
                setSwitchChecked(false);
                mSwitch.setEnabled(true);
                break;
        }
    }
    private void setSwitchChecked(boolean checked) {
        if (checked != mSwitch.isChecked()) {
            mStateMachineEvent = true;
            mSwitch.setChecked(checked);
            mStateMachineEvent = false;
        }
    }
    private void handleStateChanged(
      @SuppressWarnings("unused")NetworkInfo.DetailedState state) {
        // After the refactoring from a CheckBoxPreference to a Switch,
        // this method is useless since
        // there is nowhere to display a summary.
        // This code is kept in case a future change re-introduces an associated text.
        /*
        // WifiInfo is valid if and only if Wi-Fi is enabled.
        // Here we use the state of the switch as an optimization.
        if (state!=null && mSwitch.isChecked()) {
            WifiInfo info = mWifiManager.getConnectionInfo();
```

```
            if (info != null) {
                //setSummary(Summary.get(mContext, info.getSSID(), state));
            }
        }
        */
    }
}
```

在 Android 系统的 Setting 界面中，在 wireless 配置项中会看到 Portable Wi-Fi hotspot 和 Configure Wi-Fi hotspot setting 选项，在此可以配置 AP 的名称、加密方式和密码等，如图 15-8 所示。

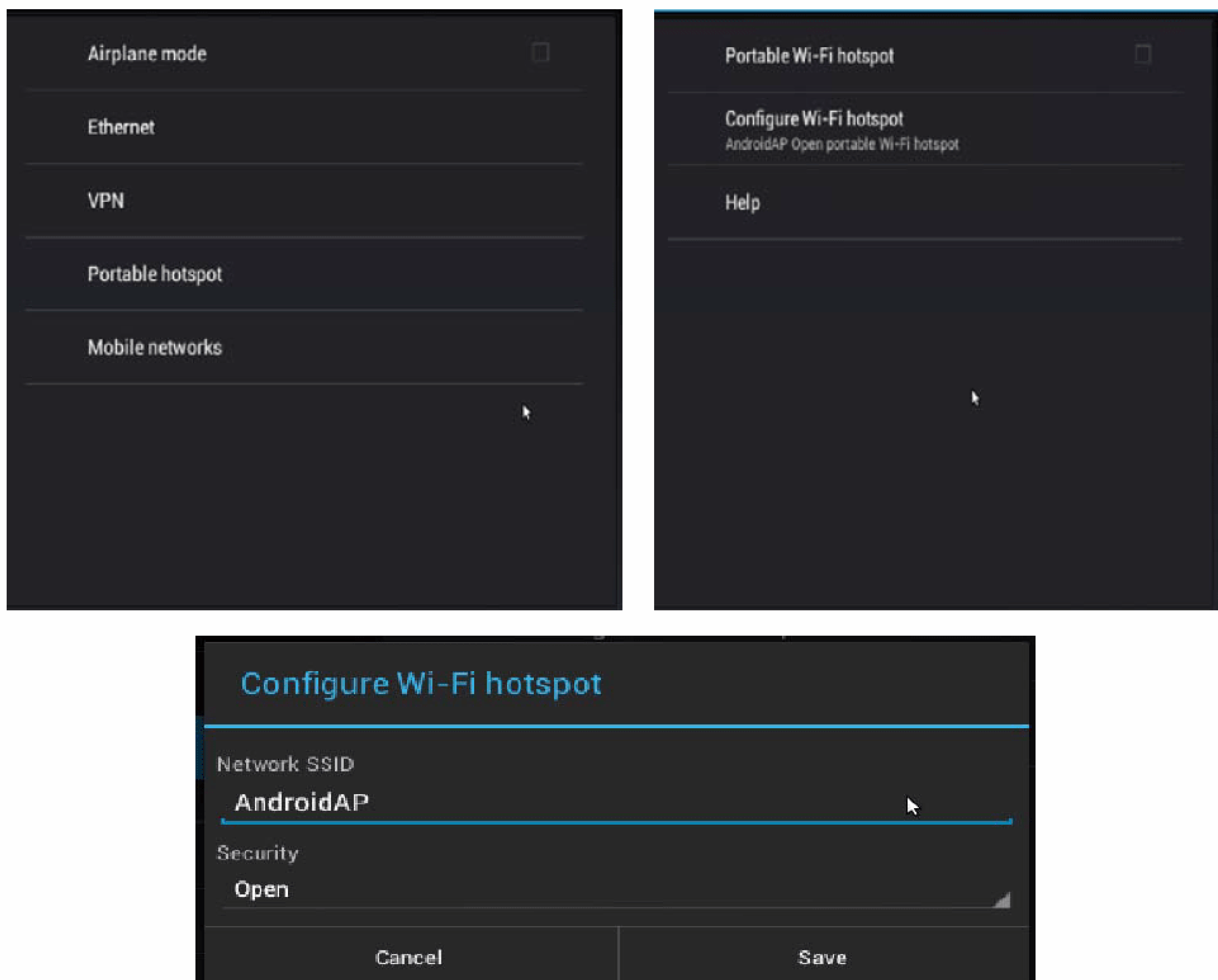


图 15-8　Portable Wi-Fi hotspot和Configure Wi-Fi hotspot setting选项

当做完上述设置后，系统开始接受响应，从此开启了整个 Android SoftAP 的启动序幕。首先通过 packages/apps/Settings/src/com/android/settings/TetherSettings.java 的 onPreferenceChange 函数接收到 Softap 状态改变信息，具体代码如下所示：

```
public boolean onPreferenceChange(Preference preference, Object value) {
    boolean enable = (Boolean)value;

    if (enable) {
        startProvisioningIfNecessary(WIFI_TETHERING);
    } else {
        mWifiApEnabler.setSoftapEnabled(false);
```

```
    }
    return false;
}
```

Softap 开启时 enable 的值为真，因而执行 startProvisioningIfNecessary(WIFI_TETHERING)：

```
private void startProvisioningIfNecessary(int choice) {
    mTetherChoice = choice;
    if (isProvisioningNeeded()) {
        Intent intent = new Intent(Intent.ACTION_MAIN);
        intent.setClassName(mProvisionApp[0], mProvisionApp[1]);
        startActivityForResult(intent, PROVISION_REQUEST);
    } else {
        startTethering();
    }
}
```

在上述代码中，isProvisioningNeeded 用来检测是否需要进行一些准备工作。如果无需准备工作，则执行 startTethering 函数，具体实现代码如下所示：

```
private void startTethering() {
    switch (mTetherChoice) {
        case WIFI_TETHERING:
            mWifiApEnabler.setSoftapEnabled(true);
            break;
        case BLUETOOTH_TETHERING:
            // turn on Bluetooth first
            BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
            if (adapter.getState() == BluetoothAdapter.STATE_OFF) {
                mBluetoothEnableForTether = true;
                adapter.enable();
                mBluetoothTether.setSummary(R.string.bluetooth_turning_on);
                mBluetoothTether.setEnabled(false);
            } else {
                BluetoothPan bluetoothPan = mBluetoothPan.get();
                if (bluetoothPan != null)
                    bluetoothPan.setBluetoothTethering(true);
                mBluetoothTether.setSummary(
                  R.string.bluetooth_tethering_available_subtext);
            }
            break;
        case USB_TETHERING:
            setUsbTethering(true);
            break;
        default:
            //should not happen
            break;
    }
}
```

在上述代码中，因为 mTetherChoice==WIFI_TETHERING 成立，所以，继而执行文件 WiFiApEnable.java 中的 setSoftapEnabled(true)函数，具体实现代码如下所示：

```
public void setSoftapEnabled(boolean enable) {
    final ContentResolver cr = mContext.getContentResolver();
    /**
    * Disable Wifi if enabling tethering
    */
    int wifiState = mWifiManager.getWifiState();
      //获取当前 wifi 的状态，如果开启，则关闭且保存状态信息到变量中
    if (enable && ((wifiState==WifiManager.WIFI_STATE_ENABLING)
      || (wifiState==WifiManager.WIFI_STATE_ENABLED))) {
        mWifiManager.setWifiEnabled(false);
        Settings.Global.putInt(cr, Settings.Global.WIFI_SAVED_STATE, 1);
    }
    if (mWifiManager.setWifiApEnabled(null, enable)) {
        /* Disable here, enabled on receiving success broadcast */
        mCheckBox.setEnabled(false);
    } else {
        mCheckBox.setSummary(R.string.wifi_error);
    }

    /**
    *  If needed, restore Wifi on tether disable
    */
    if (!enable) {
        int wifiSavedState = 0;
        try {
            wifiSavedState =
              Settings.Global.getInt(cr, Settings.Global.WIFI_SAVED_STATE);
        } catch (Settings.SettingNotFoundException e) {
            ;
        }
        if (wifiSavedState == 1) {
            mWifiManager.setWifiEnabled(true);
            Settings.Global.putInt(cr, Settings.Global.WIFI_SAVED_STATE, 0);
        }
    }
}
```

在上述代码中，首先检测 Wi-Fi 的当前状态，如果正在打开或者已经打开，则关闭 Wi-Fi 并将此状态记录下来，以便关闭 softap 时它能自动恢复到先前打开 Wi-Fi 的状态。在此调用如下文件中的 mWifiManager.setWifiApEnabled(null, enable)函数：

```
frameworks/base/wifi/java/android/net/wifi/WifiManager.java
```

具体实现代码如下所示：

```
public boolean setWifiApEnabled(WifiConfiguration wifiConfig, boolean enabled) {
    try {
```

```
        mService.setWifiApEnabled(wifiConfig, enabled);
        return true;
    } catch (RemoteException e) {
        return false;
    }
}
```

然后转向服务层的如下文件中：

```
frameworks/base/services/java/com/android/server/WifiService.java
```

通过函数 setWifiApEnabled 调用最基础的和最重要的 Wi-Fi 状态机中的 setWifiApEnabled 实例，具体实现代码如下所示：

```
public void setWifiApEnabled(WifiConfiguration wifiConfig, boolean enabled) {
    enforceChangePermission();
    mWifiStateMachine.setWifiApEnabled(wifiConfig, enabled);
}
```

另外，文件 WifiStatusTest.java 用于注册以接收不同的 Intent，具体实现代码如下所示：

```
private final BroadcastReceiver mWifiStateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(WifiManager.WIFI_STATE_CHANGED_ACTION)) {
            handleWifiStateChanged(intent.getIntExtra(
              WifiManager.EXTRA_WIFI_STATE,
              WifiManager.WIFI_STATE_UNKNOWN));
        } else if (intent.getAction().equals(
          WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
            handleNetworkStateChanged(
            (NetworkInfo)intent.getParcelableExtra(
              WifiManager.EXTRA_NETWORK_INFO));
        } else if (intent.getAction().equals(
          WifiManager.SCAN_RESULTS_AVAILABLE_ACTION)) {
            handleScanResultsAvailable();
        } else if (intent.getAction().equals(
          WifiManager.SUPPLICANT_CONNECTION_CHANGE_ACTION)) {
            /* TODO: handle supplicant connection change later */
        } else if (intent.getAction().equals(
          WifiManager.SUPPLICANT_STATE_CHANGED_ACTION)) {
            handleSupplicantStateChanged(
              (SupplicantState)intent
              .getParcelableExtra(WifiManager.EXTRA_NEW_STATE),
              intent.hasExtra(WifiManager.EXTRA_SUPPLICANT_ERROR),
              intent.getIntExtra(WifiManager.EXTRA_SUPPLICANT_ERROR, 0));
        } else if (intent.getAction().equals(WifiManager.RSSI_CHANGED_ACTION)) {
            handleSignalChanged(
              intent.getIntExtra(WifiManager.EXTRA_NEW_RSSI, 0));
        } else if (intent.getAction().equals(
          WifiManager.NETWORK_IDS_CHANGED_ACTION)) {
```

```
            /* TODO: handle network id change info later */
        } else {
            Log.e(TAG, "Received an unknown Wifi Intent");
        }
    }
};
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mWifiManager = (WifiManager)getSystemService(WIFI_SERVICE);

    mWifiStateFilter = new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    mWifiStateFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.RSSI_CHANGED_ACTION);
    mWifiStateFilter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);

    registerReceiver(mWifiStateReceiver, mWifiStateFilter);

    setContentView(R.layout.wifi status test);

    updateButton = (Button)findViewById(R.id.update);
    updateButton.setOnClickListener(updateButtonHandler);

    mWifiState = (TextView)findViewById(R.id.wifi_state);
    mNetworkState = (TextView)findViewById(R.id.network_state);
    mSupplicantState = (TextView)findViewById(R.id.supplicant_state);
    mRSSI = (TextView)findViewById(R.id.rssi);
    mBSSID = (TextView)findViewById(R.id.bssid);
    mSSID = (TextView)findViewById(R.id.ssid);
    mHiddenSSID = (TextView)findViewById(R.id.hidden_ssid);
    mIPAddr = (TextView)findViewById(R.id.ipaddr);
    mMACAddr = (TextView)findViewById(R.id.macaddr);
    mNetworkId = (TextView)findViewById(R.id.networkid);
    mLinkSpeed = (TextView)findViewById(R.id.link_speed);
    mScanList = (TextView)findViewById(R.id.scan_list);

    mPingIpAddr = (TextView)findViewById(R.id.pingIpAddr);
    mPingHostname = (TextView)findViewById(R.id.pingHostname);
    mHttpClientTest = (TextView)findViewById(R.id.httpClientTest);

    pingTestButton = (Button)findViewById(R.id.ping_test);
    pingTestButton.setOnClickListener(mPingButtonHandler);
}
OnClickListener updateButtonHandler = new OnClickListener() {
    public void onClick(View v) {
        final WifiInfo wifiInfo = mWifiManager.getConnectionInfo();
```

```
        setWifiStateText(mWifiManager.getWifiState());
        mBSSID.setText(wifiInfo.getBSSID());
        mHiddenSSID.setText(String.valueOf(wifiInfo.getHiddenSSID()));
        int ipAddr = wifiInfo.getIpAddress();
        StringBuffer ipBuf = new StringBuffer();
        ipBuf.append(ipAddr  & 0xff).append('.')
          .append((ipAddr >>>= 8) & 0xff).append('.')
          .append((ipAddr >>>= 8) & 0xff).append('.')
          .append((ipAddr >>>= 8) & 0xff);

        mIPAddr.setText(ipBuf);
        mLinkSpeed.setText(String.valueOf(wifiInfo.getLinkSpeed()) + " Mbps");
        mMACAddr.setText(wifiInfo.getMacAddress());
        mNetworkId.setText(String.valueOf(wifiInfo.getNetworkId()));
        mRSSI.setText(String.valueOf(wifiInfo.getRssi()));
        mSSID.setText(wifiInfo.getSSID());

        SupplicantState supplicantState = wifiInfo.getSupplicantState();
        setSupplicantStateText(supplicantState);
    }
};

private void setSupplicantStateText(SupplicantState supplicantState) {
    if(SupplicantState.FOUR_WAY_HANDSHAKE.equals(supplicantState)) {
        mSupplicantState.setText("FOUR WAY HANDSHAKE");
    } else if(SupplicantState.ASSOCIATED.equals(supplicantState)) {
        mSupplicantState.setText("ASSOCIATED");
    } else if(SupplicantState.ASSOCIATING.equals(supplicantState)) {
        mSupplicantState.setText("ASSOCIATING");
    } else if(SupplicantState.COMPLETED.equals(supplicantState)) {
        mSupplicantState.setText("COMPLETED");
    } else if(SupplicantState.DISCONNECTED.equals(supplicantState)) {
        mSupplicantState.setText("DISCONNECTED");
    } else if(SupplicantState.DORMANT.equals(supplicantState)) {
        mSupplicantState.setText("DORMANT");
    } else if(SupplicantState.GROUP_HANDSHAKE.equals(supplicantState)) {
        mSupplicantState.setText("GROUP HANDSHAKE");
    } else if(SupplicantState.INACTIVE.equals(supplicantState)) {
        mSupplicantState.setText("INACTIVE");
    } else if(SupplicantState.INVALID.equals(supplicantState)) {
        mSupplicantState.setText("INVALID");
    } else if(SupplicantState.SCANNING.equals(supplicantState)) {
        mSupplicantState.setText("SCANNING");
    } else if(SupplicantState.UNINITIALIZED.equals(supplicantState)) {
        mSupplicantState.setText("UNINITIALIZED");
    } else {
        mSupplicantState.setText("BAD");
        Log.e(TAG, "supplicant state is bad");
    }
```

```
}
private void setWifiStateText(int wifiState) {
    String wifiStateString;
    switch(wifiState) {
        case WifiManager.WIFI_STATE_DISABLING:
            wifiStateString = getString(R.string.wifi_state_disabling);
            break;
        case WifiManager.WIFI_STATE_DISABLED:
            wifiStateString = getString(R.string.wifi_state_disabled);
            break;
        case WifiManager.WIFI_STATE_ENABLING:
            wifiStateString = getString(R.string.wifi_state_enabling);
            break;
        case WifiManager.WIFI_STATE_ENABLED:
            wifiStateString = getString(R.string.wifi_state_enabled);
            break;
        case WifiManager.WIFI_STATE_UNKNOWN:
            wifiStateString = getString(R.string.wifi_state_unknown);
            break;
        default:
            wifiStateString = "BAD";
            Log.e(TAG, "wifi state is bad");
            break;
    }
    mWifiState.setText(wifiStateString);
}
```